

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



容器即服务

从零构建企业级容器集群

林帆 / 著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn



林 帆（花名金戟）

阿里巴巴研发效能事业部技术专家。前ThoughtWorks资深DevOps技术咨询师，2015年极客邦CNut全球容器技术大会讲师，2016年CSDN架构技术实战峰会讲师，2017年StuQ容器集群技术直播课程讲师。具有丰富的一线开发和运维经验，是国内早期的容器技术实践者和布道师。

容器即服务

从零构建企业级容器集群

林帆 / 著

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书介绍了容器即服务的发展过程和主要技术,重点阐述当下主流的 SwarmKit、Kubernetes、Mesos 和 Rancher 开源容器集群方案,并探讨了容器技术在网络、存储、监控、日志等方面的运用场景和基础知识,以及该领域在近年来的一些新的发展方向。

本书适合一线架构师、开发者、运维人员以及技术管理者进行阅读。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

容器即服务:从零构建企业级容器集群 / 林帆著. —北京:电子工业出版社, 2018.4
ISBN 978-7-121-33276-0

I. ①容… II. ①林… III. ①Linux 操作系统—程序设计 IV. ①TP316.85

中国版本图书馆 CIP 数据核字 (2017) 第 309385 号

策划编辑:张春雨

责任编辑:牛 勇

印 刷:三河市鑫金马印装有限公司

装 订:三河市鑫金马印装有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本:787×980 1/16 印张:30 字数:578 千字

版 次:2018 年 4 月第 1 版

印 次:2018 年 4 月第 1 次印刷

定 价:99.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:(010) 51260888-819, faq@phei.com.cn。

序

在这个日新月异的时代，每一位站在浪尖上的技术匠人，都不得不加紧步伐，追赶不断变化的趋势。与此相应的一个现象是，当一本技术类的书籍刚刚面市，它所讲述的内容就已经开始迅速过时。

这本书从 2016 年初开始筹备，由于种种原因拖沓了近两年终于完稿。在这段时间里：

- SwarmKit 诞生了，原先的 Swarm 技术栈光荣退役。
- Kubernetes 的版本从 1.0 一路更迭到 1.10，增加了无数新特性。
- Mesos 推出 Unified Container，曾经一度被看好的 Docker 集成器风光不再。
- Rancher 发布 2.0 版本，完全颠覆先前的用户体验设计。
- Docker 自家的 LinuxKit、阿里的 Pouch 这些底层开源技术在不断演进。

书还没写完，最初准备的材料有一大半都已经作废。

先前笔者写作《CoreOS 实践之路》一书时，同样是一边增加新章节，一边关注书里涉及软件的变化，对已有章节进行三番五次的补充修正，到完成时，许多地方都被大段大段地重写了。此次的《容器即服务：从零构建企业级容器集群》因为涉及方面较多，加上写作时间跨度较大，以至于维护其中的内容变化更加困难，经过数次截稿日的跳票，才费劲地将书中示例涉及的大部分软件更新到 2017 年中下旬的版本。

不过，本书写作的初衷并非在于介绍最新的工具。对于学习一门成熟的工具，最直接的方式莫过于阅读它的文档。但面对一个领域中众多的知识，入门者最容易迷失的地方在于缺少一条主线。本书一方面希望为容器集群及其周边的领域勾勒一幅入门的蓝图，另一方面则是点出一些在文档中没有讲清但实际很容易迷惑用户的大坑小洼，对于细节和扩展的内容则以参考链接的形式提供。

如今的容器技术正在处于百花齐放的时期，当我们讨论到容器，很多时候已不是单纯地在说某种内核虚拟化技术，而是在谈服务集群、任务调度，以及 Cloud Native 和微服务。与此同时，容器平台相关的应用场景也越来越丰富，大规模容器化部署的运用逐渐从少数大型企业发展到许多中型和创业企业里。作为现代产品发布模式的重塑者，容器技术以及它所提倡的基础设施即代码交付思想，对每位一线架构师、开发者、运维人员乃至技术管

理者的工作带来的影响，都不容小觑。本书截取了一些具有当下时代特征的技术剪影，提供给读者品味。

在编写内容时，本书尽量以通用的容器技术作为背景，而非限定于特定的容器产品（比如 Docker）。但在一些具体的例子方面，均采用了当前最主流的 Docker 容器作为讲解示例。

由于写作周期较长，加之作者个人的经验所限，书中难免存在一些阐述不当和错误的地方。本书的勘误表发布在博文视点官方网站 <http://www.broadview.com.cn/33276>，恳请各位读者通过此页面提交勘误或发邮件到 linfan.china@gmail.com 予以指正。

最后，感谢在过去两年中不断督促和鼓励我完成写作的张春雨以及负责了整本书编辑的吴倩雪，没有你们的努力，这本书肯定无法按时出版。感谢将我养育成材的父母以及我的爱人杨斌清，你们默默的支持使我得以静下心来认真地完成这部作品。同样感谢每一位开源代码的贡献者，正是开源推动了技术的革命，才使“旧时王谢堂前燕”，如今“飞入寻常百姓家”。我亦是一名普通的技术匠人，且当少一些浮躁，多一些沉淀，借以此书自勉。

林 帆

2017 年 12 月 25 日

读者服务

轻松注册成为博文视点社区用户 (www.broadview.com.cn), 扫码直达本书页面。

- **提交勘误:** 您对书中内容的修改意见可在 [提交勘误](#) 处提交, 若被采纳, 将获赠博文视点社区积分 (在您购买电子书时, 积分可用来抵扣相应金额)。
- **交流互动:** 在页面下方 [读者评论](#) 处留下您的疑问或观点, 与我们和其他读者一同学习交流。

页面入口: <http://www.broadview.com.cn/33276>



目 录

第 1 部分 基础概念

第 1 章 容器集群综述	2
1.1 虚拟化与容器	2
1.1.1 计算资源虚拟化	2
1.1.2 容器技术的本质	4
1.1.3 基于容器的软件交付	13
1.2 容器集群与分布式服务	16
1.2.1 微服务架构	16
1.2.2 容器集群生态圈	18
1.3 容器即服务	26
1.3.1 从基础设施到平台	26
1.3.2 数据中心操作系统	29
1.4 本章小结	31

第 2 部分 解决方案

第 2 章 SwarmKit 集群解决方案	35
2.1 开源容器集群方案	35
2.1.1 容器社区的“四朵金花”	35
2.1.2 经典 Swarm、SwarmKit 和 Swarm Mode	36
2.2 使用 SwarmKit	37
2.2.1 SwarmKit 综述	37
2.2.2 创建 SwarmKit 集群	40

2.2.3	在 SwarmKit 集群上运行服务	43
2.2.4	SwarmKit 集群的其他功能	45
2.3	Docker Swarm Mode	45
2.3.1	Swarm Mode 综述	45
2.3.2	集群的创建与销毁	46
2.3.3	节点管理	48
2.3.4	服务管理	51
2.3.5	服务编排	56
2.3.6	应用栈的管理	63
2.3.7	外置配置和密文管理	66
2.4	Swarm Mode 的图形界面	69
2.4.1	Swarm Mode UI 现状	69
2.4.2	Portainer	71
2.5	本章小结	74
第 3 章	Kubernetes 集群解决方案	75
3.1	Kubernetes 集群概述	75
3.1.1	Kubernetes 项目的起源	75
3.1.2	Kubernetes 的结构	76
3.1.3	基本概念	78
3.2	部署 Kubernetes 集群	82
3.2.1	使用 Minikube	82
3.2.2	使用 kubeadm	83
3.2.3	理解 Kubernetes 集群的部署过程	87
3.2.4	验证集群可用性	96
3.3	使用 Kubernetes	98
3.3.1	通过 Kubernetes 部署服务	98
3.3.2	服务的在线更新和回滚	103
3.3.3	单次任务、定时任务和全局服务	109
3.3.4	持久化存储	113

3.3.5	配置存储	116
3.3.6	管理有状态的服务	122
3.3.7	健康检查	126
3.3.8	提供对外服务	127
3.3.9	多租户隔离和配额	131
3.3.10	集群的节点管理	135
3.4	Kubernetes 包管理工具 Helm	137
3.4.1	Helm 简介	137
3.4.2	使用 Helm 管理服务	137
3.4.3	自定义 Chart	142
3.4.4	Chart 仓库	146
3.5	本章小结	147
第 4 章	Mesos 集群解决方案	148
4.1	Mesos 和 DC/OS 概述	148
4.1.1	Mesos 项目的起源	148
4.1.2	Mesos 的结构	149
4.1.3	Mesos 的内部构成	151
4.1.4	DC/OS 数据中心操作系统	152
4.2	部署 Mesos 集群	153
4.2.1	部署 ZooKeeper	153
4.2.2	部署 Mesos	157
4.2.3	启动 Master 节点	158
4.2.4	添加 Agent 节点	161
4.2.5	Mesos 服务的启动参数	164
4.3	使用 Marathon 管理服务	170
4.3.1	部署 Marathon	170
4.3.2	添加一个应用	172
4.3.3	使用 DC/OS 命令行工具	177
4.3.4	使用 Docker 容器	177

4.3.5	使用 Unified Container	179
4.3.6	持久化卷存储	182
4.3.7	Marathon-LB 负载均衡	184
4.3.8	Mesos-DNS 域名服务	188
4.3.9	服务依赖和编组	191
4.3.10	应用升级	194
4.3.11	调度约束	199
4.3.12	健康检查	201
4.4	使用 Chronos	203
4.4.1	部署 Chronos	203
4.4.2	定时表达式	204
4.4.3	创建定时任务	205
4.4.4	定时任务的依赖	208
4.5	更多的 Mesos 服务框架	209
4.5.1	Mesos 服务框架的本质	209
4.5.2	编写自己的 Mesos 服务框架	211
4.5.3	其他常见服务框架	216
4.6	DC/OS	218
4.6.1	DC/OS 简介	218
4.6.2	部署 DC/OS	219
4.6.3	DC/OS 的操作	228
4.6.4	DC/OS 命令行工具	230
4.6.5	DC/OS 的应用仓库	231
4.7	本章小结	234
第 5 章	Rancher 集群解决方案	235
5.1	Rancher 集群概述	235
5.1.1	Rancher 项目的起源	235
5.1.2	Rancher 的结构	236
5.1.3	相关概念	237

5.2	构建 Rancher 集群	239
5.2.1	部署 Server 节点	239
5.2.2	Server 节点的高可用部署方式	240
5.2.3	添加 Agent 节点	241
5.3	Rancher 的服务管理	243
5.3.1	使用 Rancher Web UI 创建服务	243
5.3.2	从容器	245
5.3.3	特殊类型的服务	247
5.3.4	使用应用商店	251
5.3.5	服务编排	252
5.3.6	服务的升级和回滚	254
5.4	Rancher 使用进阶	256
5.4.1	Rancher 的标签	256
5.4.2	调度选项	257
5.4.3	服务健康检查	258
5.4.4	Rancher 的元数据服务	260
5.4.5	Rancher 的 DNS 服务	262
5.4.6	使用私有镜像仓库	263
5.4.7	Rancher 的 Secret 服务	264
5.4.8	在应用商店添加自定义应用	265
5.5	Rancher 的命令行工具	268
5.5.1	配置 Rancher 命令行工具	268
5.5.2	命令工具的基本使用	270
5.5.3	通过命令行进行服务编排	273
5.5.4	通过命令行进行服务升级	273
5.6	使用 Rancher 安装 Kubernetes	274
5.6.1	Rancher 的环境管理	274
5.6.2	在 Rancher 中添加 Kubernetes 环境	276
5.6.3	在 Rancher 中使用 Kubernetes	279

5.7 本章小结	282
----------	-----

第 3 部分 技术周边

第 6 章 容器集群的网络和存储	284
------------------	-----

6.1 容器网络	284
6.1.1 容器网络标准	284
6.1.2 本地网络	288
6.1.3 跨节点网络	293
6.1.4 使用 Docker 内置的 Overlay 类型网络	300
6.1.5 构建基于 Flannel 的覆盖网络	301
6.1.6 构建基于 Calico 的 BGP 路由网络	306
6.2 容器存储	310
6.2.1 容器实例和镜像的存储	310
6.2.2 容器卷的存储	312
6.2.3 容器卷存储标准	316
6.2.4 基于 NFS 的卷存储	317
6.2.5 基于 Ceph 的卷存储	320
6.2.6 使用公有云存储	330
6.3 本章小结	332

第 7 章 容器服务的基础设施	333
-----------------	-----

7.1 集群性能监控	333
7.1.1 常见的开源性能监控方案	333
7.1.2 基于 TICK Stack 的性能监控	335
7.1.3 TICK Stack 的部署和使用	336
7.1.4 基于 Prometheus 的性能监控	341
7.1.5 Prometheus 的部署	343
7.1.6 Prometheus 的使用	353
7.2 集群日志管理	361
7.2.1 常见的开源日志管理方案	361

7.2.2 基于 Elastic Stack 的日志管理	363
7.2.3 基于 Fluentd 的日志管理	372
7.3 服务发现	377
7.3.1 常见的服务发现方案	377
7.3.2 Etcd	379
7.3.3 Consul	390
7.4 镜像仓库	398
7.4.1 容器镜像仓库概述	398
7.4.2 Registry	399
7.4.3 Harbor	405
7.5 本章小结	412
第 8 章 容器技术新风向	413
8.1 安全的集群操作系统：Container Linux	413
8.1.1 Container Linux 概述	413
8.1.2 Container Linux 的部署	416
8.1.3 Container Linux 的使用	418
8.2 基于容器的操作系统：RancherOS	419
8.2.1 RancherOS 概述	419
8.2.2 部署 RancherOS	421
8.2.3 RancherOS 的使用	422
8.2.4 使用 ros 工具管理系统	424
8.3 容器式的虚拟机：Hyper	429
8.3.1 Hyper 概述	429
8.3.2 部署 Hyper	430
8.3.3 Hyper 的使用	431
8.4 虚拟机式的容器：LXD	434
8.4.1 LXD 概述	434
8.4.2 LXD 的安装和使用	435
8.4.3 服务热迁移	440

8.5 容器与虚拟机的统一: Rkt.....	442
8.5.1 Rkt 概述.....	442
8.5.2 Rkt 的安装和使用.....	444
8.6 企业级定制容器: Pouch.....	450
8.6.1 Pouch 概述.....	450
8.6.2 Pouch 的开源生态.....	453
8.6.3 体验 Pouch.....	455
8.7 微内核操作系统: Unikernel.....	458
8.7.1 Unikernel 概述.....	458
8.7.2 Unikernel 的发展.....	460
8.7.3 体验 Unikernel.....	462
8.8 本章小结.....	465

第1部分 基础概念

容器即服务的核心在于容器技术，它的流行与近年来大规模、分布式、无状态服务的发展趋势密切相关。容器归根到底只是一系列内核特性的组合，以及基于此的许多实践理念，这些看似简单的概念改变了软件交付的面貌，在此之上又变化出许多新颖的套路。万变不离其宗，唯有谙熟其中门道，才能在学习的过程中拨云见日、融会贯通。

▣ 容器集群综述

第 1 章 容器集群综述

容器集群并不是许多容器的简单堆积，而是以容器技术为基础的包含部署、调度、网络、存储等方面的有机整体。在容器集群之上可以构建更高层的服务系统，如动态伸缩的任务队列服务、企业级的业务平台、分布式的数据计算服务等。作为底层计算资源和上层业务服务的黏合剂，以按需使用的方式提供基于容器的云端运行环境的平台，形成了一种具有独特价值的服务，这类场景被称为容器即服务。

这一章将从虚拟化和容器说起，介绍容器集群以及容器即服务平台的一些运用场景。

1.1 虚拟化与容器

1.1.1 计算资源虚拟化

虚拟化在现代计算机领域的使用相当广泛，它通过将真实的硬件抽象为软件可控的逻辑单元，使得昂贵的硬件资源能够按需、按量分配，以达到减少浪费、实现硬件利用率最大化的目的。计算资源的虚拟化是虚拟化领域里比较重要的一个分支，这里的计算资源主要指的是 CPU、内存、硬盘等与计算机运算直接相关的硬件资源。在很长的一段时间里，计算资源的虚拟化始终是各类虚拟机技术争夺的热土。从最初 IBM 和 Sun 等公司主导的早期 CPU 虚拟化实现，到 VMware、Xen、KVM、QEMU 等虚拟化或半虚拟化技术的成熟，就经历了 40 余年的发展过程。

与此同时，另一种虚拟化技术的分支也在缓慢发展。这类虚拟化技术不依赖与硬件相关的特性，而是在系统内核的层次之上，将进程运行的上下文环境加以限制和隔离。最初它们并不被视为虚拟化方法，比如在 Unix 系统中引入的 chroot 工具仅仅是将特定进程的文件上下文锁定在特定目录中，制造出在同个系统里模拟多个隔离的系统目录的效果。随后，在 FreeBSD 4.0 中出现的 Jails 和前 SWsoft 公司（现已更名为 Parallels）开发的基于 Windows

系统的 Virtuozzo（睿拓）等技术在 chroot 的基础上增加了进程空间和网络空间的隔离，更好地实现了进程之间互不干扰地共享硬件资源的目的。这种虚拟化方式就是最早的容器技术雏形。

随后不久，IBM、Sun 和惠普等老牌虚拟机和操作系统公司也纷纷进入不依赖特定 CPU 和硬件支持的虚拟化技术阵营，分别在自家的操作系统里推出相应的产品，例如运行在 Solaris 系统的 Zones、运行在 IBM AIX 小型机系统的 WPARs 以及运行在惠普服务器系统 HP-UX 的 SRP Containers 等。在这段时期里，特别值得一提的是在开源 GNU/Linux 系统上实现的操作系统级虚拟化服务：Linux-VServer。

开源社区的介入使得这类虚拟化技术迅速发展，并被应用到更多的领域中。然而作为社区产品，由于参与人员众多、早期目标定位不明确，Linux-VServer 的配置细节很复杂，加上文档十分混乱，因此当时只有对 Linux 内核有一定了解的用户才能驾驭它。这种状态一直持续到 2005 年，这一年，曾经设计了 Virtuozzo 的 SWsoft 公司开始在 Linux 系统上开发一款全新的开源虚拟化产品：OpenVZ^①。这款采用了 GNU/GPL 协议开源的软件很好地改善了 Linux 系统上进行虚拟化隔离的使用体验，并为 SWsoft 公司带来了可观的收入。同一时期还诞生了一个对虚拟化技术影响颇远的概念：VPS（Virtual Private Server，虚拟专用服务器）。VPS 技术是指将一台服务器分割成多个逻辑上的虚拟专享服务器。每个 VPS 都可分配独立公网 IP 地址、独立操作系统、独立硬盘空间、独立内存和 CPU 资源。这种虚拟主机间的隔离服务为用户和应用程序模拟出“独占使用计算资源”的体验。OpenVZ 理所当然地成为了当时虚拟化技术的代表之一，与 Xen、KVM 并列成为 VPS 提供商首选的虚拟化实施方案。

此时的 Linux 系统级虚拟化技术已经逐渐成熟，然而对代码质量颇为严苛的 Linux 内核团队并没有采纳 Linux-Vserver 或 OpenVZ 提交的内核补丁，而是在 Linux 2.6 的内核中重新设计了 Namespace 和 CGroup 等功能，实现了更加灵活的可组合式虚拟化能力。随后在 2008 年，Linux 社区就出现了基于内核隔离能力设计的 LXC（Linux Containers）虚拟化项目，它充分利用 Linux 内核的 Namespace 隔离能力和 CGroup（Control Groups，控制组）控制能力实现了操作系统内核层面上的虚拟化，不再需要修改内核代码，大大降低了使用该项技术的门槛。此后的几年里，又相继出现了 linux-utils 和 systemd-nspawn^②等 Linux 内核虚拟化工具。Linux 系统开始在内核虚拟化技术的演进过程中发挥越来越重要的作用，如图 1-1 所示。

① <https://wiki.openvz.org/History>

② <https://www.freedesktop.org/software/systemd/man/systemd-nspawn.html>

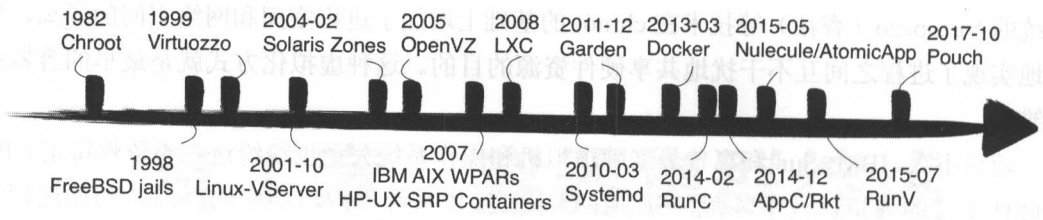


图 1-1 内核虚拟化技术的演进

2013 年 3 月，DotCloud 公司（现已更名为 Docker 公司）基于 LXC 项目封装了一套工具，将其命名为 Docker。在 Docker 的宣传下，这种基于操作系统的虚拟化技术被广泛地称为 Container（容器），并为许多开发者所知。2014 年，Docker 重新设计自己的虚拟化层并发布了 Libcontainer 项目，从此脱离了对 LXC 的依赖，成为一个独立发展的平台。2015 年，Docker 与微软、IBM 等几十家公司共同成立规范化容器技术标准的 OCI^①组织（Open Container Initiative，开放容器计划），并设立新的符合 OCI 标准的容器引擎 RunC^②。在此期间，由 CoreOS 公司主导的 Rkt 容器项目也在迅速地成长，作为比 Docker 更加轻量的容器工具成为了许多容器集群技术的备选搭配方案。

1.1.2 容器技术的本质

虚拟机和容器技术的目的都是抽象硬件并对系统资源提供隔离和配额，然而两者在本质上有着很大的差异。

虚拟机的原理是通过额外的虚拟化层，将虚拟机中运行的操作系统指令翻译成宿主机系统能够执行的系统调用，然后操作具体的硬件。这样做能够比较好地实现虚拟机和宿主机操作系统的异构，例如在 Linux 系统上运行 Windows 的虚拟机，或是在 Mac 系统上运行 Linux 的虚拟机。其缺点是通常需要依赖硬件支持，特别是 CPU 虚拟化的支持。

容器技术则完全建立在操作系统内核特性之上，是一种与运行硬件无关的虚拟化技术。由于这种方式实现的虚拟化没有转换异构指令的虚拟化层，因此在运行效率上较虚拟机方式更高，但只能实现与宿主操作系统相同系统的虚拟化。在实际使用中，有时会将容器技术和虚拟机结合，以实现“跨不同操作系统”运行容器的目的，Windows 和 Mac 版本的 Docker 就是这样的例子。

① <https://www.opencontainers.org/>
② <https://runc.io>

虚拟机和容器技术的结构差异如图 1-2 所示。注意容器结构中的“容器工具”只是一个逻辑层，它并没有与容器中的应用程序存在实际的层级关系，而是同样运行在宿主操作系统上，两者是两个不同隔离空间中的平级服务而已。

前文在介绍容器技术演进史时已经提到，现代 Linux 系统中的容器技术主要是利用内核的 Namespace 特性和 CGroup 特性实现了服务进程组的资源隔离和配额。

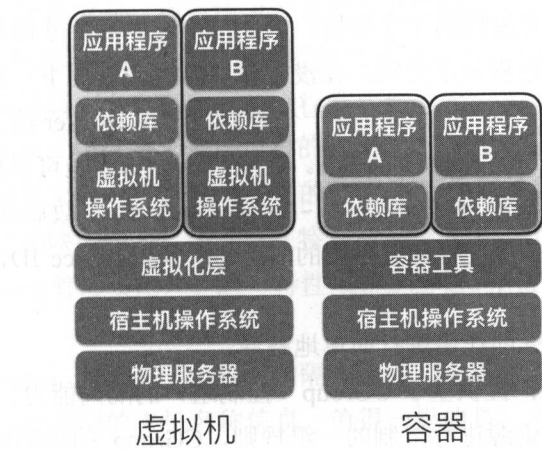


图 1-2 虚拟机与容器的对比

1. Namespace

Linux 内核实现 Namespace 的主要目的就是为了实现内核级虚拟化（容器）服务，让同一个 Namespace 下的进程可以感知彼此的变化，同时又能确保对外界的进程一无所知。这样就可以让容器中的进程产生错觉，仿佛置身于一个独立的系统环境中，以达到独立和隔离的目的。

Mount Namespace 在 2002 年进入 Linux 2.4.19 内核，它是内核中最早出现的、用于运行时隔离的 Namespace。较新的 Linux 4.7.1 版本内核已经实现了 7 种不同的 Namespace 类型，除了比较特殊的 CGroup Namespace^①用于隔离进程组对不同 CGroup 的可视性外，其他的 6 种都和某些传统意义上的系统资源直接相关。

通过查看“/proc”目录下以进程 PID 作为名称的子目录中的信息，能够了解该进程的一组 Namespace ID，如下所示。

```
# ls -l /proc/1240/ns
total 0
```

① http://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html

```
lrwxrwxrwx 1 root root 0 Aug 18 15:46 cgroup -> cgroup:[4026531835]
lrwxrwxrwx 1 root root 0 Aug 18 15:46 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 Aug 18 15:46 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0 Aug 18 15:46 net -> net:[4026531993]
lrwxrwxrwx 1 root root 0 Aug 18 15:46 pid -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 Aug 18 15:46 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Aug 18 15:46 uts -> uts:[4026531838]
```

显然，每个进程都具有这样的 7 个属性，因此每个进程都会分别在这些 Namespace 控制的系统资源上与另一些进程共享空间。在没有使用容器的情况下，系统中所有进程都具有相同的 Namespace ID 组合。然而如果有一个进程运行在 Docker 或其他容器里，它就很有可能具有完全不同的一组 Namespace ID。这些 Namespace 是可以任意组合的，容器中的进程也可以只做部分的隔离，例如在使用 `--network=host` 参数启动的 Docker 容器中运行的进程就会与主机上的其他进程拥有相同的 Network Namespace ID，此时它可以直接通过 127.0.0.1 的地址访问主机上运行的服务。

下面对每种 Namespace 的作用进行简单地解释。

- **CGroup Namespace:** 提供基于 CGroup（控制组）的隔离能力。CGroup 是 Linux 在内核级对进程可用资源进行限制的一组规则，CGroup 的隔离能够让不同进程组看到的 CGroup 规则各不相同，为不同进程组采用各自的配额标准提供便利。
- **IPC Namespace:** 提供基于 System V 进程信道的隔离能力。IPC 全称为 Inter-Process Communication，是 Linux 中一种标准的进程间通信方式，包括共享内存、信号量、消息队列等具体方法。IPC 隔离使得只有在同一个命名空间下的进程才能相互通信，这一特性对于消除不同容器空间中进程的相互影响具有十分重要的作用。
- **Mount Namespace:** 提供基于磁盘挂载点和文件系统的隔离能力。这种隔离的效果与 chroot 系统调用十分相似，但从实际原理来看，MNT Namespace 会为隔离空间创建独立的 mount 节点树，而 chroot 只改变了当前上下文的根 mount 节点位置，从而影响文件系统查找文件和目录的结果。在文件系统隔离的作用下，容器中的进程将无法访问到容器以外的任何文件。在必要情况下，可以通过挂载额外目录的方式和主机共享文件系统。
- **Network Namespace:** 提供基于网络栈的隔离能力。网络栈的隔离允许使用者将特定的网卡与特定容器中的进程运行上下文关联起来，使得同一个网卡在主机和容器中分别呈现不同的名称。Network Namespace 的重要作用之一就是让每个容器通过命名空间来隔离和管理自己的网卡配置。因此可以创建一个普通的虚拟网卡，并将它作为特定容器运行环境的默认网卡 eth0 使用。这些虚拟网络网卡最终可以通过某些

方式（NAT、VxLan、SDN 等）连接到实际的物理网卡上，从而实现像普通主机一样的网络通信。

- **PID Namespace:** 提供基于进程的隔离能力。进程隔离使得在容器中的首个进程成为所在命名空间中 PID 值为 1 的进程。在 Linux 系统中，PID 为 1 的进程地位非常特殊，它作为所有进程的根父进程，有很多特权，比如屏蔽信号、接管孤儿进程等。一个比较直观的现象是，当系统中的某个子进程脱离了父进程（例如父进程意外结束），那么它的父进程就会自动成为系统的根父进程。此外，当系统中的根父进程退出时，所有同属一个命名空间的进程都会被杀死。
- **User Namespace:** 提供基于系统用户的隔离能力。系统用户隔离是指同一个系统用户在不同的命名空间中可以拥有不同的 UID（用户标识）和 GID（组标识），它们之间存在一定的映射关系。因此，在特定命名空间中，UID 为 0 的用户并不一定是整个系统的 root 管理员用户。这一特性限制了容器的用户权限，有利于保护主机系统的安全。
- **UTS Namespace:** 提供基于主机名的隔离能力。主机名隔离是指每个独立容器空间中的程序可以有不同主机名称信息。值得一提的是，主机名只是一个用于标示虚拟主机或容器空间的代号，并不一定是全网唯一的，允许重复。因此，它虽然可以在网络中用于通信或定位服务，但并不是可靠的方法。

2. CGroup

CGroup 是 Linux 内核提供了一种可以限制、记录、隔离进程组所使用的物理资源（包括 CPU、内存、磁盘 I/O 速度等）的机制，它的 v1 版本由 Google 的 Paul Menage 设计，并在 2007 年进入 Linux 2.6.24 内核（这个内核版本实际上在 2008 年 1 月才正式发布），之后发布的 v2 版本在 2016 年 3 月已经成为了 Linux 4.5 内核的一部分。CGroup 也是 LXC 等现代容器管理虚拟化系统资源的手段。

CGroup 最初设计出来是为了统一 Linux 下混乱的资源管理工具，例如过去限制 CPU 使用时可以用 `renice` 和 `cpulimit` 命令，限制内存要用 `ulimit` 或者 PAM（Pluggable Authentication Modules），而限制磁盘 I/O 和网络又需要其他的专用工具。CGroup 作为一种内核级的资源限制手段，在功能和效率方面都是早期工具不能比拟的。值得一说的是，在现代的 Linux 进程管理中，CGroup 的使用已经比较普遍，并不是非得使用容器才与 CGroup 有关。例如在采用了 Systemd 管理服务的 Linux 发行版中，其 `service` 文件定义中使用的 `MemoryLimit`、`BlockIOWeight` 等配置其实就是在间接地为进程配置 CGroup。又如在专门为大规模服务器部署而设计的 CoreOS 操作系统里，为了避免在系统升级时因占用数据

带宽而影响正常业务服务，同样使用 CGroup 的管控实现了“只在带宽空闲时下载”的效果，这些应用都是与容器没有直接关系的。

在 Linux “一切皆文件”的思想中，CGroup 同样直观地表现为一些特殊的目录和文件。查看任意一个进程在“/proc”目录下的内容，可以看到下面这样一个名为“cgroup”的文件。

```
# cat /proc/1240/cgroup
11:perf_event:/
10:blkio:/
9:freezer:/
8:cpu,cpuacct:/
7:net_cls,net_prio:/
6:devices:/
5:cpuset:/
4:pids:/
3:memory:/
2:hugetlb:/
1:name=systemd:/
```

这个文件中除了最后一行，都对应了一个 CGroup 的子系统。每个子系统是 CGroup 中用于定义某类资源控制规则的结构。与 Namespace 类似，如果观察运行在主机上每个进程的 cgroup 文件内容，会发现它们中的绝大多数都一样，也就是说共用了相同的 CGroup。但如果观察一个运行在容器中的进程，会发现它具有与其他进程完全不同的一组 CGroup 路径。

那么这个路径到底指的是哪里呢？来看一下当前系统中的所有挂载点，会看到其中有许多是 CGroup 类型的项目，如下所示。

```
# mount | grep 'type cgroup'
cgroup on /sys/fs/cgroup/systemd type cgroup (...)
cgroup on /sys/fs/cgroup/cpuset type cgroup (...)
cgroup on /sys/fs/cgroup/blkio type cgroup (...)
cgroup on /sys/fs/cgroup/perf_event type cgroup (...)
cgroup on /sys/fs/cgroup/pids type cgroup (...)
cgroup on /sys/fs/cgroup/devices type cgroup (...)
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (...)
cgroup on /sys/fs/cgroup/memory type cgroup (...)
cgroup on /sys/fs/cgroup/freezer type cgroup (...)
```

这当中的每个挂载点都是一个 CGroup 子系统的根目录，例如上一例中那个进程所属的 cpuset 子系统路径为“/”，实际上就是指“/sys/fs/cgroup/cpuset”这个目录，其余子系统

的位置也可以此类推。

在 Linux 4.7.1 内核中，已经支持了 10 类不同的子系统，分别如下所示。

- hugetlb 子系统用于限制进程对大页内存（Hugepage）的使用。
- memory 子系统用于限制进程对内存和 Swap 的使用，并生成每个进程使用的内存资源报告。
- pids 子系统用于限制每个 CGroup 中能够创建的进程总数。
- cpuset 子系统在多核系统中为进程分配独立 CPU 和内存。
- devices 子系统可允许或者拒绝进程访问特定设备。
- net_cls 和 net_prio 子系统用于标记每个网络包，并控制网卡优先级。
- cpu 和 cpuacct 子系统用于限制进程对 CPU 的用量，并生成每个进程所使用的 CPU 报告。
- freezer 子系统可以挂起或者恢复特定的进程。
- blkio 子系统用于为进程对块设备（比如磁盘、USB 等）限制输入 / 输出。
- perf_event 子系统可以监测属于特定的 CGroup 的所有线程以及运行在特定 CPU 上的线程。

为了比较方便地与系统 CGroup 进行交互，可以安装 CGroupTools 工具包。对于 Debian 或 Ubuntu 系统可使用 `apt-get` 或 `apt` 命令安装，如下所示。

```
# apt install cgroup-tools
```

Redhat 或 CentOS 系统则可用 `yum` 或 `dnf` 命令来安装，如下所示。

```
# dnf install libcgroup-tools.x86_64
```

这个工具包中包含了一组用于创建和修改 CGroup 信息的命令。以通过 CGroup 实现 CPU 的配额限制为例，安装完 `cgroup-tools` 工具包后，通过 `cgcreate` 命令来创建两个 CPU CGroup 的子系统分组（需要 root 权限来执行），如下所示。

```
# cgcreate -g cpu:/cpu50
# cgcreate -g cpu:/cpu30
```

其中 `-g cpu` 表示设定的是 CPU 子系统的配额。除 CPU 子系统外，`cgcreate` 同样可以创建其他各种配额的子系统。通过 `lscgroup` 命令就能看到创建出来的这两个子系统，如下所示。

```
# lscgroup
... ..
cpu:/
```



```
cpu:/cpu30
cpu:/cpu50
... ..
```

接下来为两个 CPU 分组分别设置一条限制规则，CPU 子系统的 `cfs_quota_us`^①可以设定进程在每个“时间片周期”内可占用的最大 CPU 时间，单位是 μs 。CPU 时间片周期由子系统的 `cfs_period_us` 属性指定，默认为 100000，单位同样是 μs 。因此例如 `cfs_quota_us` 的数值 50000 和 30000 表示在该组中的进程最多分别能够使用 50%和 30% 的 CPU 时间。使用 `cgset` 命令将规则设定进去，如下所示。

```
# cgset -r cpu.cfs_quota_us=50000 cpu50
# cgset -r cpu.cfs_quota_us=30000 cpu30
```

构造一个耗 CPU 的程序，例如下面这个不断累加并输出数据的脚本，将其命名为“app.sh”。

```
#!/bin/bash
N=0
while true; do
    N=$((N+1))
    echo $N
done
```

首先，尝试在后台直接运行它，如下所示

```
# ./app.sh > /dev/null &
```

接着在一个单核的机器上进行测试。观察系统 CPU 的使用情况，用不了多久，这个进程就会将所有 CPU 资源统统耗尽，如下所示。

```
# ps aux
USER      PID    %CPU  %MEM    .....  COMMAND
root      10366  99.6   0.0    .....  /bin/bash ./app.sh
```

然而如果使用 `cgexec` 命令让 `app.sh` 进程在 `cpu50` 这个 CGroup 中重新运行，就会发现这次进程的 CPU 使用被稳定地限制在了 50%附近，如下所示。

```
# cgexec -g cpu:cpu50 ./app.sh > /dev/null &
# ps aux
USER      PID    %CPU  %MEM    .....  COMMAND
root      10366  51.8   0.0    .....  /bin/bash ./app.sh
```

① https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-cpu

再启动一个相同的进程，同样放到 `cpu50` 子系统里。此时两个进程的 CPU 使用率都有所降低，其总和依然大约为 50%，如下所示。

```
# cgexec -g cpu:cpu50 ./app.sh > /dev/null &
# ps aux
USER      PID    %CPU  %MEM    .....
```

USER	PID	%CPU	%MEM	COMMAND
root	10380	32.2	0.0	/bin/bash ./app.sh
root	10384	24.6	0.0	/bin/bash ./app.sh

现在不妨在这个子系统里再启动一个相同的进程实例，然后观察 CPU 使用率的变化，可以发现 CGroup 的效果的确是作用于同属于该子系统中所有进程的，如下所示。

```
# ps aux
USER      PID    %CPU  %MEM    .....
```

USER	PID	%CPU	%MEM	COMMAND
root	10380	18.6	0.0	/bin/bash ./app.sh
root	10384	17.5	0.0	/bin/bash ./app.sh
root	10390	16.5	0.0	/bin/bash ./app.sh

如果在刚才创建的 `cpu30` 子系统中重复这个测试，则所有进程的总 CPU 用量将稳定在 30% 左右，如下所示。

```
# cgexec -g cpu:cpu30 ./app.sh > /dev/null &
# cgexec -g cpu:cpu30 ./app.sh > /dev/null &
# cgexec -g cpu:cpu30 ./app.sh > /dev/null &
# ps aux
USER      PID    %CPU  %MEM    .....
```

USER	PID	%CPU	%MEM	COMMAND
root	10410	14.3	0.0	/bin/bash ./app.sh
root	10433	10.5	0.0	/bin/bash ./app.sh
root	10450	9.9	0.0	/bin/bash ./app.sh

本质来说，对 CGroup 的所有操作都是在对系统挂载的 CGroup 目录进行修改。上述操作实际是在 `/sys/fs/cgroup/cpu/` 目录下创建了 `cpu50` 和 `cpu30` 这两个子目录，查看其中任何一个目录的内容，就会看到类似下面这样的文件结构。

```
# ls /sys/fs/cgroup/cpu/cpu30/
cgroup.clone_children
cgroup.event_control
cgroup.procs
cpu.cfs_period_us
cpu.cfs_quota_us
cpu.shares
cpu.stat
notify_on_release
tasks
```

观察里面的几个文件内容，如下所示，基本上就真相大白了。

```
# cat /sys/fs/cgroup/cpu/cpu30/cpu.cfs_quota_us
30000
# cat /sys/fs/cgroup/cpu/cpu30/tasks
10410
10433
10450
```

实际完全可以不通过 `cgroup-tools` 工具来完成以上的所有操作，只要用普通的 Linux 命令创建出这个目录结构就可以达到相同的目的。例如下面的命令创建了一个限制 CPU 使用率为 20% 的子系统。

```
# mkdir /sys/fs/cgroup/cpu/cpu20/
# echo 20000 > /sys/fs/cgroup/cpu/cpu20/cpu.cfs_quota_us
# ./app.sh > /dev/null &
# echo $! >> /sys/fs/cgroup/cpu/cpu20/tasks
```

稍微解释一下上面的命令。首先，由于“`/sys/fs/cgroup/cpu`”目录是被挂载为 CGroup 类型的文件系统，当用户在该目录创建子目录时，系统会自动在该目录创建作为 CGroup 子系统所需的文件结构，因此在后续的操作中就可以直接读写这些文件了。其次，在最后一条命令中的 `$!` 是 Bash 中用于获取前一个命令 PID 的方式，因此这个命令的意思是将前一条命令执行的 `app.sh` 进程 PID 写到子系统的 `tasks` 文件里面。

不难看出，CGroup 的文件结构至少包含三层目录树，它的三个核心概念及相互间的逻辑关系如下所示。

- **Hierarchy (控制树)**: 每个控制树表示系统中挂载的一套 CGroup 配置树，其中可以包含多个子系统。
- **Subsystem (子系统)**: 每个子系统对应一种控制资源，比如 CPU 子系统就是控制 CPU 时间分配的一个控制器。
- **Control Group (控制组)**: 在每个子系统下都可以创建不同的控制组，而每个控制组可以被赋予一组进程，通过指定控制组的参数来限制该控制组的进程对相应系统资源的使用。

这样的体系实际上形成了一种层级状的 CGroup 控制链，如图 1-3 所示。

当然，以上只是一个很简略的例子，在实际情况中，容器使用的 CGroup 约束方式会更加复杂，关于 Namespace 和 CGroup 的细节这里就不再展开了。

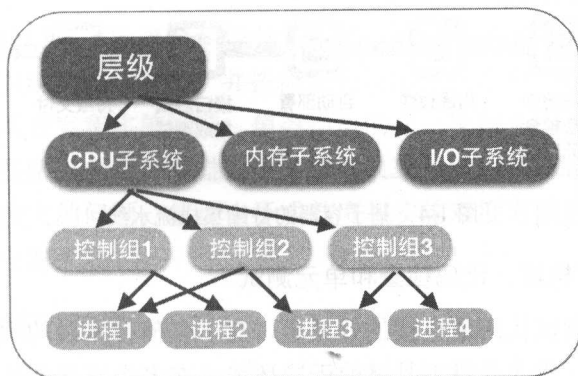


图 1-3 CGroup 的层级控制树

1.1.3 基于容器的软件交付

容器技术在内核中出现的时间要远远早于 Docker 和 Rkt 这种容器管理软件。然而不可否认的是，Docker 项目是迄今为止最成功的容器技术产品之一。Docker 的成功并非因为它在技术领域有什么重大突破，它的最初版本只是 LXC 工具的一层封装，Docker 的独特之处在于它引入了镜像、版本、仓库等一系列的思想，以及 Build、Ship、Run 这样的软件交付理念，从而彻底地改变了软件发布的过程。

在软件开发实践中，CI (Continuous Integration, 持续集成) 和 CD (Continuous Delivery, 持续交付) 是经常被用来提高模块间集成测试频率和优化发布流程的手段，通过可视化交付流程、监控代码变化、自动化部署和运行测试，以此更加频繁地获得软件质量反馈。此外，流水线还能很好地反映实际产品交付过程中所经历的各个环节，尤其是只要将这些环节涉及的实践自动化，就能看到从代码提交到每个阶段的测试、部署过程中潜在的流程问题和交付瓶颈。

容器技术会直接改变软件打包、发布和运维的许多方面，这些实践通过持续集成 / 交付的流水线就能十分直观地体现出来。图 1-4 展示了一个典型的使用容器交付的项目的持续集成流水线，末尾省略号的部分表示流程包含的其他步骤，直到最终部署上线。在这个环环相扣的链条里，容器的运用场景贯穿全程。

具体来说，这些影响体现在以下几个方面。

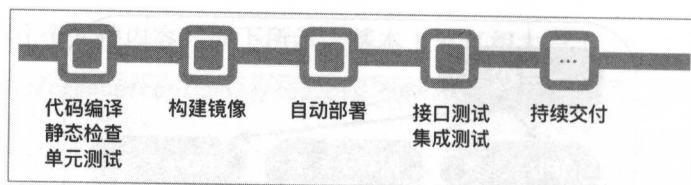


图 1-4 基于容器的持续集成流水线

1. 在容器中进行构建、代码检查和单元测试

在容器中构建和测试代码最直接的好处是，对于任何技术栈的项目，总是能够恰到好处地提供一个适合运行相应开发工具链的干净环境。在多个不同产品项目需要采用不同版本、种类的构建工具或 SDK 的时候，这个优势特别有用，干净的执行环境能够减少构建或测试结果与预期不一致的情况。此外，由于有些代码检测（如安全性扫描）、全量的集成测试和回归测试需要运行较长时间，通常放在夜间执行，而白天进行的构建任务则相对较多，容器除了能够分别提供所需的环境外，还具有更好地利用容器动态扩缩的特性，很适合混合调度这些周期性的任务，将基础设施资源池化。

根据所用的编程语言和技术栈，可将与构建和单元测试相关的镜像分为很多种类，常用的 Docker 镜像包括以下这些。

- C/C++: https://hub.docker.com/r/_/gcc/。
- Java: https://hub.docker.com/r/_/openjdk/。
- Golang: https://hub.docker.com/r/_/golang/。
- Nodejs: https://hub.docker.com/r/_/node/。
- 其他语言的基础镜像。

在实际使用时往往需要基于这些镜像添加相应的框架和 SDK 库，以满足使用的要求。

2. 通过镜像打包和分发服务

容器的封装意味着，不论运行的服务是用 Java、Python、PHP 还是 Golang 设计的，平台都可以用几乎相同的方式去完成部署，而不用考虑安装服务所需的环境，容器能够做到这一点的原因就在于它采用了镜像打包和分发方式。

以 Docker 容器为例，所有的运行时依赖都会在服务设计的时候以声明或描述的方式体现在一个 Dockerfile 文件中。当服务打包时，将依次执行其中的描述性代码，使得服务以及所需的完整执行环境，包括文件目录、环境变量、启动参数等全部固化在镜像里。这种工作方式将最终运行环境的定义提前到了编码设计的阶段，缓解了开发和运维之间的信息不对称问题，对促进团队 DevOps 文化起到了积极的作用。

镜像仓库是当下使用最多的容器镜像分发方式，通过一个集中式的存储点，将构建出的镜像按照一定的组织结构进行管理，并按需发布到运行服务的节点上。本书的第7章还会讨论容器仓库的技术方案和实施细节。值得指出的是，镜像仓库并不是唯一的分发容器打包产物的方法，一些研究项目也在尝试通过例如共享存储、P2P 传输等途径解决镜像仓库中心节点流量压力较大的问题，然而由于成熟度和实施难度方面的原因，这些研究并没有被作为主流的镜像分发手段。

3. 使用容器优化测试过程

在测试的过程中，需要依赖的外部系统或组件往往比较多，然而这些系统或组件未必总是处于空闲或可用状态。这时就需要有大量的临时资源或是模拟资源（称为“桩”或“Mock”）。几种典型的测试依赖资源包括浏览器、数据库和第三方服务接口。对于这些场景，容器都有用武之地。

临时的浏览器资源主要用于界面测试。由于每个浏览器在执行测试时一般都由某个用例独占，为了并发地完成界面测试，测试人员需要准备许多不同种类的浏览器资源。特别在自动化并行测试的时候，对浏览器种类和数量的需求变化是十分迅速的，必须按照实际用量的最大值来准备这些资源，因而造成不少浪费。此外，目前进行与浏览器相关的测试通常都是使用 Selenium/WebDriver 和相关的 SDK 来驱动的，提前部署、长期维护所需的服务和相应驱动同样都很费时。

容器提供了很好的临时浏览器提供方法。实际上，Selenium 提供了一组基于 Chrome 和 Firefox 并预装了相应驱动的浏览器镜像。该项目的 GitHub 地址为：<https://github.com/SeleniumHQ/docker-selenium>。

对于测试时期的数据库依赖，比如与数据相关的集成测试，由于测试过程需要反复运行，如果前一次测试运行没有正常地结束，或是没有正确地清理留下的数据，就特别容易影响后续测试的运行结果。容器恰恰是提供这种即用即弃基础设施的最佳方式，完全可以在测试脚本中先启动一个全新的 MySQL 服务，测试完就销毁，保证了每次测试的独立性。

在 Docker Hub 上有许多主流数据库的镜像可供使用。例如 MySQL 和 MongoDB 的镜像地址分别是 https://hub.docker.com/r/_/mysql 和 https://hub.docker.com/r/_/mongo。

这些镜像都提供了在容器启动时修改数据库配置和注入初始化数据的方法。例如 MySQL 的镜像启动后会自动执行放在“/docker-entrypoint-initdb.d”目录下的所有 SQL 或 Shell 脚本，而 MongoDB 的镜像则建议用户将初始化过的数据内容直接通过 Volume 挂载到容器的“/data/db”目录下。

更加常见的场景是测试时需要通过“服务桩”来模拟某些第三方服务的行为，开源社

区里已经有许多很不错的解决方法。其中获得过 Oracle 颁发的 Duke 选择奖的 Moco^①就是一款值得采用的工具。

虽然 Moco 的使用本来就十分简单，在某些集群测试的场景下，将它容器化仍然能获得快速分发和运行环境无关性的好处。有些社区镜像已经提供了这方面的功能，例如 Docker Hub 上的 <https://hub.docker.com/r/rezzza/docker-moco> 这个镜像。

使用者仅仅需要将一个被模拟接口的配置描述文件挂载到该容器的特定位置，就实现了具备特定功能的“服务桩”。

4. 通过容器进行集群自动化部署和管理

部署意味着产品的最终发布上线。在传统流程中，软件的部署有时是十分耗时、费力的事情，特别是线上环境的部署，一点小小的差错就会造成严重的后果。在用传统的虚拟机方式部署服务时，相应的操作通常都会使用一个经过反复优化的自动化脚本来完成，但这些复杂的脚本依然可能在一些环节上出现问题，容器化方式的部署通过从根本上简化部署的过程，可以达到提升部署可靠性的目的。

对于单个服务而言，通过容器进行部署只需要两个动作，先是停止旧镜像运行的服务，然后使用新的镜像启动服务。在集群中进行服务的部署，虽然本质上与简单地部署单个容器没多少差别，但加上在实际运用时需要涉及的因素，就会复杂很多。

像 Docker 这样的容器，现在已经具有了许多成熟、开源的集群调度和管理工具，例如 SwarmKit、Kubernetes、Mesos 或 Rancher，它们的价值在于自动地完成了节点选择、网络配置、路由切换等一系列的额外工作。同时由于容器使服务不再依赖于主机的配置和环境，从整体来看，采用容器在集群中部署服务依然比采用虚拟机的方式更容易管控。此外，容器化的集群工具通常还具有任务编排和扩展 API 等能力，能够简化日常的运维和比较方便地进行二次开发。

1.2 容器集群与分布式服务

1.2.1 微服务架构

近年来容器技术的繁荣同样得益于大规模分布式软件架构的推波助澜。随着现代软件

^① <https://github.com/dreamhead/moco>

规模不断扩大，单纯地增加节点内存和运算能力已经不足以应对业务请求的需要，由于能够利用服务数量的横向扩展来获得更好的业务性能和承载量，分布式架构成为了许多企业选择的探索方向。

微服务架构（Microservices Architecture）是在 Martin Fowler 和 James Lewis 合著的一篇博客^①里正式提出的一种软件架构形式，它是分布式架构在互联网时代发展过程中的全新诠释。微服务架构提倡将应用分割成一系列细小的服务，每个服务专注于单一业务功能，运行于独立的进程中，服务之间边界清晰，采用轻量级通信机制（如 HTTP/REST）相互沟通，配合起来又能实现完整的应用，满足业务和用户的需求。

康威定律（Conway's law）指出：“任何设计系统的组织，最终产生的设计等同于组织之内、之间的沟通结构”。在传统软件开发项目中，通常将开发者按技能分为前端层、中间层、数据层。这种项目设计架构的分层结构对应了不同团队的人员组织结构：前端对应的角色为网页开发和设计人员；中间层对应的角色为服务端业务和开发人员；数据层对应着 DBA 等角色。所有的软件层需要最终被集成在一起，才能成为一个可部署的整体。而微服务架构的开发模式则将项目按业务能力划分为不同的服务，每个服务都要求在对应业务领域的全栈（从前端到后端）软件中实现，从界面到数据存储再到外部沟通协作等。这样，原本单一的服务就按照功能边界被分解成一系列独立、专注的微服务。每个微服务对应传统项目中的一个组件，但是可以独立编译、测试和部署。

相较于分层架构的项目，微服务架构具备以下优势。

1. 复杂度可控

在将项目分解的同时，微服务架构规避了复杂度无止境积累的恶性趋势。每一个微服务专注于单一功能，并通过定义良好的接口来清晰地表述服务边界。由于体积小、复杂度低，每个微服务可由一个小规模开发团队完全掌控，易于保持高可维护性和开发效率。

2. 独立部署

由于微服务具备独立的运行进程，所以每个微服务也可以独立部署。当某个微服务发生变更时无须编译、部署整个项目。由微服务组成的项目相当于具备一系列可并行的发布流程，使得发布更加高效，同时可降低对生产环境所造成的风险，最终实现缩短项目交付周期。这十分契合互联网 SaaS 类型服务每天都要发布很多次的常态需求。

^① <http://martinfowler.com/articles/microservices.html>

3. 技术选型灵活

微服务架构中的技术选型是去中心化的。每个团队可以根据自身服务的需求和行业发展的现状，自由选择最适合的技术栈。由于每个微服务相对简单，因此对技术栈进行升级时所面临的风险较低，甚至可以完全重构一个微服务。这使得对于特定领域的功能，可以充分利用该领域中最好的框架或编程语言来解决问题，而不受系统中其他服务的技术选型制约。

4. 容错和故障隔离

当某一组件发生故障时，在单一进程的传统架构下，故障很有可能在进程内扩散，形成系统全局性的不可用。在微服务架构下，故障会被隔离在单个服务中。若设计良好，其他服务可通过重试、平稳退化等机制实现系统层面的容错。

5. 灵活的扩展性

分层开发的整体系统并非一定不能进行横向扩展，只是对于大型系统而言，以整个系统为单位进行复制扩展的代价相对较高。事实上系统中网络流量和计算资源的需求差异往往是与业务功能挂钩的，例如报表计算的业务可能需要大量 CPU，而购票的业务系统则需要更多的入口带宽，此时微服务架构的灵活性就体现出来了，因为每个服务可以根据实际需求独立进行扩展。

然而，这样的架构要是在早些年提出来，也许还要面临着项目运维的极大挑战。微服务架构项目中的每个独立服务团队都是自己构建、发布和维护自己的分布式微服务应用的，出于技术栈和工具选择的差异，完整部署这样一套系统就需要所有服务团队集体参与，因为很难有人能够熟悉每一个微服务的部署细节。容器技术则为微服务理念提供了统一的打包、部署和状态监控的方式，成为这个架构实施不至于失控的十分关键的环节。

微服务架构本质上是一种分布式的系统，在部署结构上与集群关系十分紧密。容器技术在集群上的运用已经比较成熟，特别是在服务调度和资源分配方面，这使得服务在设计完成后能够独立、快速地完成上线。

1.2.2 容器集群生态圈

容器集群技术的发展并不是孤立的单点，而是伴随着虚拟化技术、容器引擎、编排技术、操作系统、容器仓库、监控技术等周边技术链的共同繁荣。

图 1-5 是容器集群所处的完整生态。

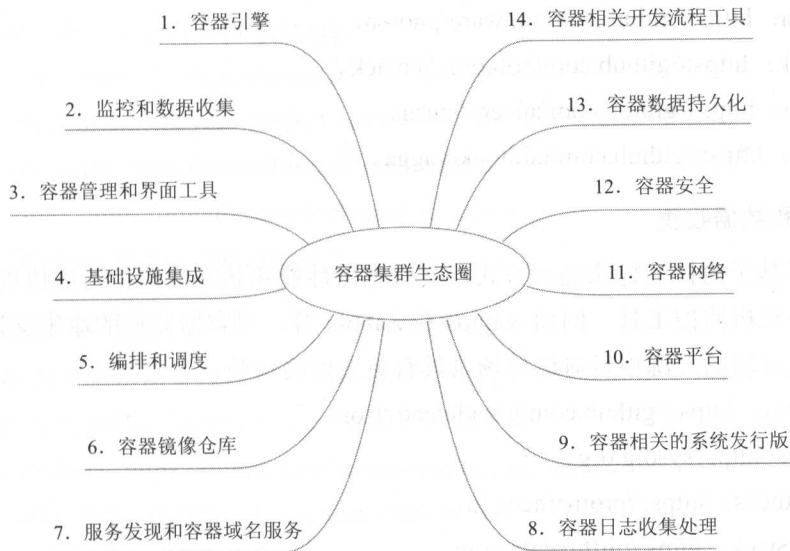


图 1-5 容器集群生态圈

1. 容器引擎

容器引擎是容器集群生态圈的核心部分，它是与内核 Namespace 和 CGroup 等功能直接交互，并提供相应 API 使得外部能够与之集成的工具或服务。Docker 无疑是目前为止设计最成功、使用最广泛的容器引擎之一。在 1.12 版本以后，Docker 的容器化功能实际上已经由独立的项目 RunC 来实现，但 Docker 项目依然作为一种集成完备的容器引擎工具得到许多用户的喜爱。下面列举一些开源的容器引擎。

- Docker: <https://www.docker.com>。
- Rkt: <https://coreos.com/rkt>。
- Pouch: <https://github.com/alibaba/pouch>。
- Systemd-nspawn: <https://www.freedesktop.org/wiki/Software/systemd>。
- Hyper: <https://hyper.sh>。^①
- Garden: <https://github.com/cloudfoundry/garden>。
- LXC: <https://linuxcontainers.org>。

① 从严格意义上看，Hyper 采用虚拟机的方式对环境进行隔离，并不是一种基于容器的隔离方案，但它能很好地与 Docker 或 Kubernetes 等容器集群技术相结合，取代其环境隔离的功能，因此也归属此列。本书的第 8 章将进一步对 Hyper 进行介绍。

- Photon: <https://github.com/vmware/photon>。
- Jetpack: <https://github.com/3ofcoins/jetpack>。
- Kurma: <https://github.com/apcera/kurma>。
- Vagga: <https://github.com/tailhook/vagga>。^①

2. 监控和数据收集

由于容器基于内核的特殊隔离方式，其对容器性能和状态的监控与虚拟机存在一些差别。传统的虚拟机监控工具，例如 Nagios 和 Zabbix 等，对容器监控的原生支持并不完善。而以下一些新启动的开源项目对这种场景具有更友好的体验。

- cAdvisor: <https://github.com/google/cadvisor>。
- Sysdig: <http://sysdig.org>。
- Prometheus: <https://prometheus.io>。
- TICK-Stack: <https://influxdata.com>。
- Docker-Alertd: <https://github.com/deltaskelta/docker-alertd>。
- Grafana: <https://grafana.com>。

其中的 TICK-Stack 指的是 Influxdata 推出的 Telegraf、InfluxDB、Chronograf、Kapacitor 四款开源工具，不过自 TICK-Stack 推出 1.0 版本以后，这些工具分别提供了开源版本和企业版本，后者增加了例如高可用、云端存储等企业级功能。

3. 容器管理和界面工具

可视化是用户友好性十分重要的一种体现，Shipyard 和 Decking 是 Docker 早期十分受欢迎的可视化工具，之后 Docker 又收购了 Kitematic 作为官方的容器管理 UI。但随着容器应用集群化，早期的 UI 工具不再流行，一些针对特定集群平台定制的新型管理 UI 开始出现。例如 Kubernetes 官方推出的 Dashboard 项目可用于可视化地管理集群，Cockpit 则是红帽公司推出的 Kubernetes 集群管理界面。以下是一些开源的容器管理 UI 项目。

- Kitematic: <https://kitematic.com>。
- DockerUI: <https://github.com/crosbymichael/dockerui>。
- Panamax: <http://panamax.io>。
- Rapid Dashboard: <https://github.com/ozlerhakan/rapid>。

^① 这些项目只是社区中众多的支持不同平台和具备不同特性的容器引擎的冰山一角，Google 曾经主导的 lmcftfy (<http://lmcftfy.io/>) 项目就是一个十分优秀的容器引擎，可惜的是该项目自 2015 年就不再维护了。

- Cockpit: <http://cockpit-project.org>。
- Portainer: <https://www.portainer.io>。
- Shipyard: <http://shipyard-project.com>。
- Seagull: <https://github.com/tobegit3hub/seagull>。
- Dockeron: <https://github.com/dockeron/dockeron>。
- DockStation: <https://dockstation.io>。

4. 基础设施集成

容器集群的实施是需要以硬件基础设施作为依托的,有些辅助工具能够简化这个过程。这些项目往往与具体的底层平台相关,如下所示。

- Nova-docker: <https://github.com/stackforge/nova-docker>。
- Magnum: <https://github.com/openstack/magnum>。
- Machine: <https://docs.docker.com/machine>。
- Boot2Docker: <https://github.com/boot2docker/boot2docker>。
- Clocker: <https://github.com/brooklyncentral/clocker>。
- MaestroNG: <https://github.com/signalfuse/maestro-ng>。

Nova-docker 和 Magnum 都是在 OpenStack 中集成容器集群的项目。不过目前 OpenStack 官方正在尝试通过让 Kubernetes 直接创建虚拟机的方式来统一它在 IaaS 层和 CaaS 层的差异,其中的 Nova-docker 已经被废弃了。Machine 是 Docker 公司推出的基础设施管理工具。Boot2Docker 曾经是在 Windows 和 Mac 上使用 Docker 的官方方案,但随着 Docker 1.12 版本发布了多种操作系统的发行版后,已经不再被推荐使用。

5. 编排和调度

编排和调度是容器集群的基本功能,因此选择编排和调度工具实际上就是在选择容器集群的方案。以下是一些开源的容器任务编排调度工具。

- SwarmKit: <https://github.com/docker/swarmkit>。
- Kubernetes: <http://kubernetes.io>。
- Marathon: <https://github.com/mesosphere/marathon>。
- Rancher: <http://www.rancher.com>。
- Nomad: <https://github.com/hashicorp/nomad>。
- OpenShift: <https://www.openshift.com>。
- Crane: <https://github.com/michaelsauter/crane>。

- Nebula: <https://github.com/nebula-orchestrator>。
- GearD: <http://openshift.github.io/geard>。

其中的 OpenShift 主要是指其 3.0 之后的发行版（早期版本未开源），它是红帽公司基于 Kubernetes 二次开发的集持续集成和交付于一体的容器集群方案，它具有开源和商业两个版本。

6. 容器镜像仓库

镜像仓库是基于容器的、在软件发布流程中必要的组成部分，Docker 开源了其镜像仓库的最小实现，但对于企业级应用来说，它缺少了高可用、权限控制、管理界面等必要功能。Docker Hub 和国内的许多容器云平台都提供了公有云的企业级仓库服务，社区中也有一些容器仓库的开源实现，如下所示。

- Repository: <https://github.com/docker/distribution>。
- Nexus: <http://www.sonatype.org/nexus>。
- Harbor: <http://vmware.github.io/harbor>。
- Portus: <https://github.com/SUSE/Portus>。
- Docker Registry UI: <https://github.com/atcol/docker-registry-ui>。

其中的 Nexus 是一种通用的软件包仓库解决方案，支持包括 Maven、NPM、PIP、RPM 等许多主流打包格式的分发和管理，它在 3.0 以后的版本才开始支持作为 Docker 镜像仓库。Harbor 是 VMware 推出的企业级开源 Docker 仓库解决方案。

7. 服务发现和容器域名服务

服务发现和域名服务实际上是微服务架构和容器集群的调度工具所需的组件，它们在容器集群中十分常见，也是这个生态圈中举足轻重的一部分，以下是其中一些在实际工程中被提及较多的工具。

- Etcd: <https://github.com/coreos/etcd>。
- Consul: <http://www.consul.io>。
- ZooKeeper: [https:// ZooKeeper.apache.org](https://ZooKeeper.apache.org)。
- Eureka: <https://github.com/Netflix/eureka>。
- Traefik: <https://traefik.io>。
- Muguet: <https://github.com/mattallty/muguet>。
- Registrator: <https://github.com/gliderlabs/registrator>。
- SkyDNS: <https://github.com/skynetservices/skydns>。

8. 容器日志收集处理

和容器集群的监控类似，收集容器中的服务运行日志，与在虚拟机中的实现同样存在许多差异。目前 Docker 已经内置支持的日志收集工具包括 Splunk、Fluentd、Graylog 和 AWS、GCE 等主流云厂商的日志平台。一些过去用于虚拟机的日志收集器，比如 LogStash 或 Flume，同样能够使用容器中的服务，只是它们都不再是首选的方案。但 Elastic 公司新推出的 Beats 项目同样值得关注。以下是其中一些工具的链接。

- Splunk: <https://www.splunk.com>。
- Fluentd: <https://www.fluentd.org>。
- ElasticSearch: <https://www.elastic.co/products/elasticsearch>。
- Kibana: <https://www.elastic.co/products/kibana>。
- Beats: <https://www.elastic.co/products/beats>。
- LogStash: <https://www.elastic.co/products/logstash>。
- Flume: <https://flume.apache.org>。

ElasticSearch 和 Kibana 是开源日志索引和展示的主流选择，因此也属于与日志相关的工具。值得指出的是，Splunk 并不是开源或免费的，但它在企业级日志处理方案中的应用十分广泛。

9. 与容器相关的系统发行版

有些 Linux 发行版是为容器运行而优化的，Atomic 和 ClearLinux 系统都属于此类。另一些 Linux 发行版在设计之初就充分地将容器机制融入了系统的架构理念，例如 CoreOS。有的系统甚至将 Docker 作为系统的核心服务来管理其他用户进程，例如 RancherOS 和 Hyper 容器引擎所使用的操作系统。类似的项目还有许多，它们都是架设容器集群时十分称手的基础设施，如下所示。

- Container Linux: <http://coreos.com>。
- Project Atomic: <http://www.projectatomic.io>。
- RancherOS: <http://rancher.com/rancher-os>。
- ClearLinux: <https://clearlinux.org>。
- Photon OS: <https://vmware.github.io/photon>。
- CargoOS: <https://cargos.io>。
- SmartOS: <https://www.joyent.com/smartos>。

第8章将进一步介绍 Container Linux 和 RancherOS 这两个操作系统的细节。

10. 容器平台

容器平台是大规模容器运用的产物，它通常会与持续集成、持续交付的工具结合，连接上层应用服务和底层基础设施，帮助使用者快速实现从代码提交到产品上线全过程的端到端交付过程。这些平台继承并发扬了 PaaS（Platform as a Service）服务的定位，并由此衍生出 CaaS（Container as a Service，容器即服务）的概念。以下是其中一些相关的开源项目。

- Deis: <https://deis.com>。
- Flynn: <http://flynn.io>。
- Dokku: <https://github.com/progrium/dokku>。
- Fabric8: <http://fabric8.io>。
- Kel: <http://www.kelproject.com>。
- Nanobox: <https://nanobox.io>。
- Tsuru: <https://tsuru.io>。

除了这些开源的容器平台服务实现之外，互联网上还有许多在线按资源使用量付费的 CaaS 平台，它们也是整个容器集群生态的一部分。

11. 容器网络

容器技术在解决环境隔离和配额问题的同时，也引入了网络层面的复杂性。由于使用了 Network Namespace，每个容器都可以获得独立的 IP 地址，这对于单个主机的情况并无大碍，但对于容器集群的情况，IP 地址的分配和互联就成为了新的问题。因此在设计容器集群时，通常需要针对网络的连接方式加以特殊考虑。常用的开源方案有以下这些。

- Libnetwork: <https://github.com/docker/libnetwork>。
- Flannel: <https://github.com/coreos/flannel>。
- Calico: <http://www.projectcalico.org>。
- Weave: <https://github.com/zettio/weave>。
- Romana: <http://romana.io>。
- Canal: <https://github.com/projectcalico/canal>。
- Open vSwitch: <http://openvswitch.org>。
- Pipework: <https://github.com/jpetazzo/pipework>。

这些网络方案大多采用了七层网络的 Overlay Network 方式，也就是在容器之间通信的网络包上封装了用于路由寻址的额外包头，这种方式会导致网络通信效率的下降，具体影响程度与所封装的额外数据大小有关。而 Calico 采用修改每个主机节点上的 IPTables 和路由表规则来实现容器间数据路由和访问控制，属于三层网络的方式，这种方案在节点规模

不太大（最多几百个节点）时的效率优势十分明显，是一种比较受推荐的容器网络工具。除了这些较常用的方案，一些条件允许的企业也会结合 MacVLAN 等二层网络方案实现容器的互联，以获得更好的网络性能。

12. 容器安全

容器安全性问题的根源在于容器和宿主机共用内核，因此受攻击的可能性特别大。另外，如果容器里的应用导致 Linux 内核崩溃，整个宿主机系统都会崩溃，这一点与虚拟机是不同的。此外，镜像的安全性也是容器安全的一部分，如何保障用户下载的镜像是可信的、未被篡改过的，以及如何保证镜像中不会意外包含具有大量漏洞的老旧软件，都是需要考虑的问题。目前这些安全课题主要在一些企业级应用中引起较多重视，下面是一些相关的开源工具和项目。

- Notary: <https://github.com/docker/notary>。
- Clair: <https://github.com/coreos/clair>。
- AppArmor: <https://en.wikipedia.org/wiki/AppArmor>。
- SELinux: <https://selinuxproject.org>。
- Twistlock: <https://www.twistlock.com>。
- OpenSCAP: <https://github.com/OpenSCAP/container-compliance>。

13. 容器数据持久化

容器是一种不可变的基础设施，容器的数据应该通过 Volume 的方式保存到外部的介质上，容器持久化存储本质上就是要解决如何简便地将外部存储挂载到容器中使用的问题。Docker 在其 1.9 版本后提供了存储的插件，这也为许多存储方案提供了便利，以下列举几个例子。

- Flocker: <https://github.com/clusterhq/flocker>。
- Convoy: <https://github.com/rancher/convoy>。
- REX-Ray: <https://github.com/codedellemc/rexray>。
- Netshare: <https://github.com/ContainX/docker-volume-netshare>。
- OpenStorage: <https://github.com/libopenstorage/openstorage>。

其中的 Ceph 是通用的网络存储工具，它对容器化存储具有良好的支持。

14. 容器相关开发流程工具

容器的镜像可以被看作一种新型的应用打包方式，因此容器常常与软件的开发和持续集成、持续交付流程相结合，提供在不同环境中的一致性部署能力。以下是一些利用容器

改善软件开发和交付的工具或平台。

- Drone.io: <https://drone.io>。
- Shippable: <http://shippable.com>。
- Cyclone: <https://github.com/caicloud/cyclone>。
- Screwdriver: <http://screwdriver.cd>。
- WatchTower: <https://github.com/v2tec/watchtower>。
- Wercker: <http://wercker.com>。
- Totem: <http://totem.github.io>

1.3 容器即服务

1.3.1 从基础设施到平台

在容器集群的生态圈中，各个部分相互依赖和关联，形成了如图 1-6 所示的复杂的技术栈。中间部分是作为核心的容器引擎，它需要基于特定的存储引擎和操作系统的支持。为了能够大规模地运行应用服务，在容器引擎之上还有其他各种需求，比如容器网络、服务发现、负载均衡、任务调度等，这些需求本身是容器引擎无法满足的，因此需要将各种基础设施进行组合。构建这样的一个体系后，实际上容器集群已演化成一个平台化的服务。

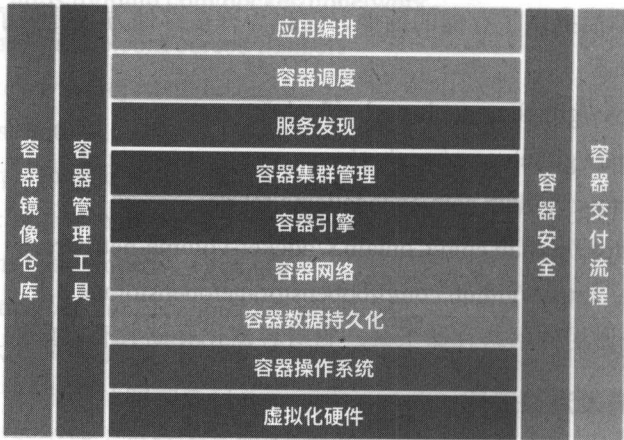


图 1-6 容器集群平台

云计算通常被划分为三层，分别是 SaaS（软件即服务）、PaaS（平台即服务）和 IaaS（基础设施即服务），它们对应了云计算的三种应用方式。SaaS 层的应用主要以与业务直接相关的服务为中心，即各行各业的互联网相关业务，如电子商务、金融保险、电信通讯等，这些不同的服务都可以通过企业应用的 SaaS 方式进行交付，从而获得动态扩缩和弹性调度的能力，而这些能力恰恰是由下层的 PaaS 提供的。云计算的发展使得大规模服务部署的需求越发旺盛，而 PaaS 和 IaaS 分别提供了这些部署的能力和载体。IaaS 以资源为中心，提供资源弹性，让整个资源以按需获取的方式提供出来。PaaS 以上层应用为中心，提供部署的弹性。

云计算的三层发展并不是十分平衡的，在最初的一段时间里，相对 IaaS 和 SaaS 而言，PaaS 的发展一直比较滞后。其根本原因在于，早期 PaaS 始终没有解决好本身的复杂度以及与应用层的耦合度问题。这个问题促使了以容器平台技术为核心的新一代 PaaS 体系的形成，为了与早期的 PaaS 区别开来，这一代平台体系通常被称为 CaaS。

用户的业务需求是动态变化的，用户需要以快速、高效的方式实现其容器应用的横向扩展。在 CaaS 中的许多特性和传统 PaaS 都有相似的地方，例如屏蔽底层资源差异、屏蔽分布式部署细节、资源动态扩缩、内置监控、日志、网络解决方案等。但从另一方面来说，CaaS 的生态是基于容器和微服务理念打造的，相较于传统 PaaS，它更加轻量，同时也具有更好的普适性，更易于维护。

可用性通常是分布式应用的重要指标。单点失效是分布式系统的基本假设，当单个容器失效时需要有机制将容器重建甚至迁移到其他计算资源上，从而保持整体服务不受影响。CaaS 也要对计算资源故障进行容错。值得一说的是，在对服务器和服务器集群进行管理的时候，有两种截然不同的管理方式，它们通常被比喻成“饲养宠物”和“放牧牲畜”，如图 1-7 所示。

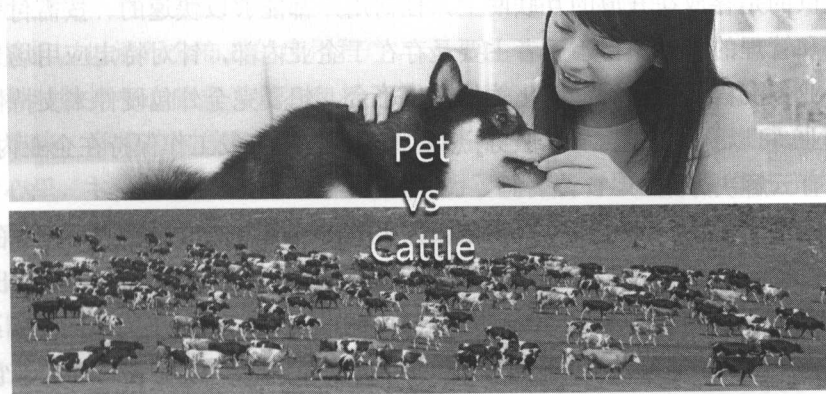


图 1-7 “饲养宠物”和“放牧牲畜”

在现实场景中，“饲养宠物”型的服务器管理方式其实十分常见。比如对于数据库服务器而言，一般企业里其绝对不允许宕机，因为数据库一旦宕机，所有服务就全部不可用了，所以要人为维护，这也就是 DBA 要做的事情。人围着机器转，这是典型的饲养宠物型的服务器管理思路，一旦数据库宕机马上需要有人去修。那么“放牧牲畜”是一种怎样的管理方式呢？对于规模巨大的集群，比如由数千、上万乃至更多服务器组成的集群，如果要确保每一个节点永远处于工作状态，所需的运维将极其复杂，人力成本将是一个天文数字。因此，集群管理的关注点不应该放在每个单独的节点上，而是一定要能够允许一些服务器处于故障状态，也就是说集群的容错能力才是最需要被关注的设计要素。实际上，对运行在一个带健康检查的负载均衡背后的数个 Web 服务来说，即使出现了一些异常，集群依然能够对外提供服务。就像是牛群中忽然少了几头牛，整个牛群依然井然有序，并不会因此进入混乱状态。

在集群中管理服务，实际上总是隐含着对服务高可用性的需求，而这两种服务的管理方式并非存在着孰优孰劣。像是数据库这类需要持久化存储服务的高可用实施方式，通常是采用多个副本和分片来实现的，虽然能够容忍一定程度的故障，但在故障发生时，人为地进行分析和恢复的确是必要和安全的做法。而像大多数 Web 应用那样不带数据状态的服务，当个别节点不可用时，只要能够快速识别出来，然后部署新的节点来替代它们就足够了，这样的过程是完全可以被自动化的。

容器本身的环境隔离和快速部署特性，对于“放牧牲畜”这样的任务来说，实在是再合适不过了。通过 CaaS 管理服务能够在系统出现软硬件故障时具有较好的容错性，放养型的运作管理方式也能够有效降低集群和 CaaS 平台本身的维护成本、运维成本。

作为云计算中一种平台式的概念，CaaS 同样存在“公有 CaaS”和“私有 CaaS”的分别。前者提供的是存放在开放的互联网上，任何用户都能够以快速的、按需付费的方式进行容器部署和管理的服务平台。后者主要是存在于企业内部，针对特定应用场景优化或是与企业自身业务流程相吻合的容器化软件交付平台。想要完全外包硬件来支持快速应用程序开发的企业，可以利用公共 CaaS 服务；想要将应用程序开发工作保持在企业内部的企业，可以利用私有云解决方案。

从技术的实现来看，CaaS 服务是需要依托于 IaaS 的基础设施实现的。服务的扩缩本质上依然是主机资源的伸缩，只是通过容器的包装屏蔽了复杂性。因此许多目前比较成功的 CaaS 平台也都是依托于特定 IaaS 实现的，下面给出了一些示例。

- Amazon ECS：全称是 Amazon EC2 Container Service，这是亚马逊基于它的云平台设计的用于直接运行容器的云端服务。

- CoreOS Tectonic: 这是 CoreOS 公司与 Google 公司合作的公有云和私有云的 CaaS 解决方案, 主要卖点在于备受推崇的 Google-style infrastructure 服务和 CoreOS 系统本身的安全特性。
- Docker Datacenter: 这是 Docker 公司推出的私有云 CaaS 解决方案, 包含 Universal Control Plane 和 Trusted Registry 等服务, 作为 Docker 官方的首选推荐, 具有不错的竞争力。
- Google Container Engine: 这是由 Google 公司提供, 架设在 Google Compute Engine 之上的公有云容器运行平台, 与亚马逊的 ECS 是直接的竞争对手。
- Project Magnum: 这是基于 OpenStack 架构的 CaaS 解决方案, 它通过 OpenStack 的 API 服务将基于 Swarm、Kubernetes、Mesos 的容器发布过程集成到私有 IaaS 云中。

国内近年来也涌现出许多耳熟能详的 CaaS 平台, 灵雀云、道客云、时速云、有容云、精灵云、希云等本土企业都相继发布了自己的容器管理系统, 笔者相信这个领域依然具有十分广阔的发展空间。

1.3.2 数据中心操作系统

虚拟化技术自诞生以后就不断地改变着人们使用硬件资源的方式。近年来, 从搜索引擎到社交网络再到微服务和 SaaS, 虚拟化一直作为支持软件规模化运用背后的关键驱动力。如果没有虚拟化所带来的服务器利用率的提升和相关成本的节约, 那么现在所使用的大多数线上公共服务以及企业级的云计算都是不可能实现的。随着数据中心转型, 虚拟化背后最初的设想, 即把一台大型、昂贵的服务器划分为多台虚拟机的理念, 已然有了不同的含义。相反, 虚拟化不再是分配个别服务器的资源, 而是将大量的服务器合并为一个仓库规模的虚拟计算机, 以运行分布式的应用程序。

维基百科对数据中心的定义是: 包括计算机系统和其他与之配套的设备, 还包含冗余的数据通信连接、环境控制设备、监控设备以及各种安全装置的一整套复杂的设施。在传统的数据中心里, 主机与任务的分配是静态的, 为了确保所有服务能获得充分的峰值资源, 通常会为每种任务分配足量的资源, 这使得主机的利用率十分低。同时, 还需要由许多 IT 专业人员组成的团队来保障所有的服务器能运行, 并且确保其中的各个应用程序都能按需获得与之匹配的网络、存储、CPU 等资源。这个工作量十分繁重, 以至于必须有一种特殊的管理系统来自动化地处理所有设备上的重复事务, 然而这些管理系统往往是专用于特定数据中心甚至是特定服务类型任务的。

数据中心操作系统是一个很有趣的概念，它的目的是设计一种通用的服务管理系统，使得用户能够像使用个人计算机操作系统那样，直观且高效地管理部署在整个数据中心的数以万计的应用服务。这个概念是加州大学伯克利分校博士生 Matei Zaharia 在 2011 年美国俄勒冈州波特兰 Usenix 年度技术会议论文 *The Datacenter Needs an Operating System*^①中提出的。事实上，Zaharia 也是著名大数据计算框架 Spark 的主要开发者。

在这篇论文里，Zaharia 认为数据中心托管了许多种类的应用程序，包括存储系统、网络应用、长期运行的服务和批量分析。并且随着计算机集群用户数量的增长，这些应用程序的数量在未来还将持续增加。许多企业都设计了自行管理系统来统一处理数据中心中运行的任务。比如，Google 利用 Pregel（一种用于图片应用的框架）、Dremel（一种用于交互式数据挖掘的低延迟系统）和 Percolator（一种增量索引系统）来增强其 MapReduce 计算能力；Facebook 用 Hadoop 数据库处理数百个用户的几乎同时互动的 SQL 查询。但是由于许多类似这些任务的管理系统都是由特定框架和体系管理的，它们之间通常无法共享计算和存储资源。因此，有必要推出一个更强大的、能够处理整个数据中心事务的通用操作系统，使得数据中心的操作人员能够在多个应用程序间有效地反复利用资源。

然而设计一个能够处理数据中心全部资源的操作系统要比制作一个用于个人设备运行应用服务的操作系统困难得多。数据中心操作系统要能够把所有这些资源集中在一个管理平台，并且提供资源共享、数据共享、编程抽象和调试，为整个数据中心提供分布式调度与协调功能，统一协调各类资源，实现数据中心级的弹性伸缩能力。其实从现代云计算的观点来看，数据中心操作系统就相当于一套完善的 IaaS+PaaS 且支持多租户的体系，但现实中存在各式各样的服务和应用，传统的 PaaS 平台对服务设计的侵入性较强，无法直接适配复杂的业务场景，需要对服务进行改造。因此当前比较成功的数据中心操作系统，包括 Google 的 Borg 和 Zaharia 设计出的 Mesos，实际上都采用了 IaaS+CaaS 的方式来实现。Borg 和 Mesos 等早期数据中心操作系统的设计都远远早于 Docker，它们各自实现了类似的容器化功能。对于所有的数据中心，无论是公共、私人还是混合模式，未来都应该趋向于采用这些统一化部署和管理的超大规模架构。这些通过智能软件、容器技术和微服务整合在一起的新型数据中心，将给企业计算带来全新的云经济和云规模，带来全新且前所未有的业务模式。

值得一提的是，“DC/OS”（Data Center Operation System）这个词已经被 Mesosphere 公司注册，成为了旗下一款数据中心操作系统产品的名称。然而从本质的定义来看，本书

① https://www.usenix.org/events/hotcloud11/tech/final_files/Zaharia.pdf

将介绍的 SwarmKit、Kubernetes、Mesos 和 Rancher 等集群方案，都实现了对容器集群的规模、生命周期和资源调度等维度的管理，因此也在不同程度上实现了数据中心操作系统的这个理念。

1.4 本章小结

容器技术的发展不是一蹴而就的，然而近年来随着 Docker 提出的 build、ship、run 理念，容器的应用为软件的交付和管理带来了显著的效率提升。

容器的集群化使用是容器发展的下一个阶段，“容器即服务”概念的提出，将容器集群本身作为了一种屏蔽服务器资源细节、提供部署和调度能力的服务。在短短几年里，围绕容器和集群发展起来的技术已经十分繁荣，逐渐形成一个至底而上的完整生态圈。本书在接下来的章节中将依次介绍几种主流的容器集群以及容器集群周边的技术方案。

第 2 部分 解决方案

工欲善其事，必先利其器。大规模的服务集群并非服务容器的简单堆积，而是一项涉及资源分配、任务调度、故障自愈、租户隔离、权限管控、网络集成等诸多方面的系统性工程。在这个领域里，开源社区已经大致形成了 SwarmKit、Kubernetes、Mesos、Rancher 四分天下的局面，它们各自汇聚一方生态，形成独具特色的整体业务解决方案。

四种方案要解决的基本都是同类问题，主要的区别只是来自于不同社区背景，应对场景的偏重性和出发点略有所不同，最终殊途同归。下表体现了这四者关键的差异。

	SwarmKit	Kubernetes	Mesos	Rancher (Cattle)
项目发起者	Docker 公司	Google 公司	加州大学伯克利分校 AMP 实验室	Rancher Labs 公司
当前维护者	Docker 公司	CNCF 基金会	Mesosphere 公司	Rancher Labs 公司
主要特点	与 Docker 无缝集成，部署和学习成本低；内置跨节点网络，功能简单且高内聚	灵活的组件设计，丰富的 API，易于扩展和定制；丰富的文档和社区资源	两层调度设计，自定义空间大；支持混合调度容器服务和传统的计算任务	可通插件开发，具备一定的扩展能力；可通过插件支持定时任务；支持多租户隔离
容器运行时	Docker (Containerd)	Docker、Rkt、Hyperd	Docker、Mesos 容器	Docker
网络插件标准	CNM	CNI	CNI	CNI
高可用机制	内置 Raft 协议选举	外置 Etcd 组件选举	外置 Zookeeper 组件选举	外置数据库和负载均衡
服务配置和密钥托管	支持	支持	无	无
用户权限管控	无	ABAC、RBAC	ABAC	ABAC、RBAC
租户隔离	无	支持动态配置	静态配置	支持动态配置
扩展方式	插件	插件、组件、资源对象	插件、任务调度器	插件
部署有状态服务	无	支持	支持	无
集群定时任务	无	支持	支持	通过插件支持
易用性	★★★	★★	★★	★★★
功能丰富性	★★	★★★★	★★★★	★★★★
社区活跃度	★★★★	★★★★	★★	★★★★
可定制性	★★	★★★★	★★★★	★★

➤ SwarmKit 集群解决方案

➤ Kubernetes 集群解决方案

➤ Mesos 集群解决方案

➤ Rancher 集群解决方案

第2章 SwarmKit 集群解决方案

在 Docker 容器的早期版本中，SwarmKit 仅仅是一种服务运行环境的隔离工具，随着其技术生态的日益完善，Docker 开始内置对网络、存储、服务编排等集群功能的支持。SwarmKit^①是伴随着 Docker 1.12 的诞生而成名的一个容器集群项目，现在已经被集成在 Docker 中，彻底地改变了 Docker 早期设计对集群不友好的形象。这个过程并不是一蹴而就的，本章将从 SwarmKit 的前身项目 Swarm^②开始，逐步介绍 SwarmKit 以及使用内置在 Docker 中的 Swarm Mode 来创建和管理集群的方法。

2.1 开源容器集群方案

2.1.1 容器社区的“四朵金花”

技术社区从来不缺乏各种各样的轮子，容器的圈子里也一度百花齐放地诞生过许多辅助规模化容器运维的工具和平台。且不说企业自研和未开源的项目，单是在 GitHub 上就能轻松找到数百种工具，功能从简单的批量部署脚本到设计精良的全功能平台，让人目不暇接。纵观这些开源项目，核心解决的问题都是如何高效地进行容器集群的资源调度和任务编排，其中 SwarmKit、Kubernetes、Mesos 和 Rancher 是目前在这个领域中最知名且使用者数量最庞大的四种全功能解决方案。

接下来将用连续的四个章节（包括本章）依次讨论这四种主流容器的部署、使用和适用场景。值得一提的是，虽然这些容器集群管理项目能够独立提供十分相似的功能，但它们之间并非是完全互斥而毫无交集的。例如，Mesos 社区曾多次尝试将 Swarm 或 Kubernetes 作为其管辖的数据中心中的一种扩展任务调度框架，Rancher 则能够快速构建和管理

① <https://github.com/docker/swarmkit>

② <https://github.com/docker/swarm>

Swarm、Kubernetes 和 Mesos 集群环境，并通过 Web UI 统一监控在这些节点上运行的容器资源。

对初识容器集群应用的用户来说，SwarmKit 是很好的起点。一方面它与 Docker 深度集成，能够充分运用许多用户已经熟悉的 Docker 概念和周边设施，学习曲线相对平缓。另一方面，它的部署结构简单紧凑，内置配置存储、服务域名路由、跨容器网络等集群基础能力，因此几乎不会因操作复杂而让人感到信心受挫。

2.1.2 经典 Swarm、SwarmKit 和 Swarm Mode

SwarmKit 并不是 Docker 公司在集群方式运用方面的第一次尝试。早在 2014 年年底，Docker 公司就着手设计一组容器集群的组合方案：Machine、Swarm 和 Compose。在这个方案中，Machine 是一款用于各种主流虚拟机和云平台快速创建 Docker 运行环境的工具，它支持在创建出来的节点上自动部署 Swarm。当时的 Swarm 是一款整合跨节点网络的集群式容器运维管理服务，它利用 Docker 守护进程的 API，将多节点的计算资源进行汇总，并提供完全兼容 Docker 的运行 API，使用者只需在执行 Docker 命令工具时，用 `--host` 参数将目标设置为 Swarm 服务的 IP 和端口，即可像操作单个节点的 Docker 服务一样操作整个容器集群。但这种方式具有先天的局限性，比如在单节点 Docker 中并没有为服务高可用而设计的副本集和负载均衡等概念，也不存在服务网络管理和跨节点数据存储的问题。此外，集群中的服务间关系和启动顺序编排也远比单节点时复杂，这些功能过去是由 Compose 组件完成的，而它所支持的定义项对于集群的场景也一度显得捉襟见肘，直到其 v2 版本 API 推出之后才逐渐完善。

在一年多后的 2016 年 2 月，Docker 公司低调地开始了 SwarmKit 项目。这个项目的作用与 Swarm 相似，但也许是意识到了 Docker 的 API 和服务模型与集群化场景的不匹配，SwarmKit 从一开始就重新设计了独立的一套 API 和模型体系，并且采用了独立的客户端命令行工具：`swarmctl`。在 SwarmKit 里，集群的节点管理被设计成了 API 中十分自然的一部分，而不再是像 Machine 这样的单独组件。同时，SwarmKit 将过去 Swarm 所依赖的外部集群一致性存储组件 Etcd 的核心部分内置到了软件中，这样虽然让使用者失去了选择存储组件的自由，却大大地简化了集群的部署过程。本书在第 7 章中会单独介绍 Etcd 这个项目。

真正让 SwarmKit 项目为众人所知的事件是 Docker 1.12 版本开始提供了一个一键创建容器集群的新命令：`docker swarm`。事实上，在这个后来被称为 Swarm Mode 的功能背后，正是 SwarmKit 的功劳。Docker 将 SwarmKit 的核心模块内嵌在了 Docker 后台服务中。Swarm Mode 并非一种特别的运行“模式”，而是 Docker 中的一组与集群相关功能的统称，因此并

不存在像“模式切换”这样的概念，Docker 通过不同的命令允许使用者同时以“当前节点”和“整个集群”两种视角来操作容器。

在 Docker Daemon 服务默认启动时，使用者只能用与过去的 Docker 命令相同的操作在当前节点上启动和管理容器，与集群相关的命令处于禁用的状态。直到使用者通过 `docker swarm` 命令从当前节点创建出新的集群，或是将当前节点加入到一个已经存在的集群里，就解封了 Swarm Mode 带来的一系列全新的命令集，包括 `docker service`、`docker stack`、`docker node`、`docker config` 和 `docker secret` 等，本章稍后会详细地介绍它们。

图 2-1 表示了 Swarm、SwarmKit 和 Swarm Mode 以及其他相关项目之间的关系。图中重叠的部分表示相应的项目之间存在代码层面的相互引用或组件形式的依赖。

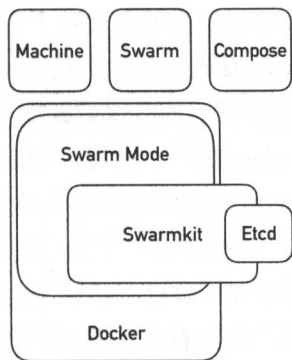


图 2-1 SwarmKit 和相关项目的关系

Swarm Mode 所创建的集群，本质上就是 SwarmKit 的集群，二者在代码实现层面上其实是同样的事物。但相比于 Swarm Mode 这个名称，SwarmKit 一词更具有识别性，因此社区里有时会用“SwarmKit 集群”来指代 Swarm Mode 所创建的集群，以便与过去的经典 Swarm 集群划清界限，本章的标题“SwarmKit 集群解决方案”就采用了这样的表述方式。本章的内容会严格区分“Swarm Mode”和“SwarmKit”的名称使用，以避免产生混淆。

2.2 使用 SwarmKit

2.2.1 SwarmKit 综述

SwarmKit 是 Docker 现代集群的基础，它的代码开源在 GitHub 的独立仓库中，主要提

供了基于 Docker 容器构建集群并进行节点管理和 Service 管理的能力。

在节点管理方面, SwarmKit 能够将普通的虚拟机赋予集群角色, 并承担节点增加、退出以及故障失联时自动处理的职责, 确保在高可用的 SwarmKit 集群里任意一个节点故障都不会影响集群的整体功能。这个能力得益于在 SwarmKit 中使用了 Etcd 项目的 Raft 模块^①。

Raft 是一种分布式一致性协议，目的在于解决在不可信任的网络集群中进行状态确认的问题。Raft 协议把集群中的节点分为三种角色：Leader、Follower、Candidate。

在任何时刻，集群中只存在一个 Leader 角色的节点，它保存整个集群的完整状态，其余节点都作为 Follower 角色，从 Leader 节点同步状态信息。在集群刚刚创建时，第一个进入集群的节点通常会将自已标记为 Leader，此后进入的节点都是 Follower。这样的角色划分并没有什么奇特之处，不过 Raft 协议的高明之处在于，集群中的角色并非是一成不变的。Leader 节点依靠定时向所有 Follower 发送心跳数据包来保持其地位，当 Follower 节点在一定的周期内没有收到来自 Leader 节点的心跳数据包时（通常是发生了网络分区或是节点故障），就会发起新一轮“Leader 选举”。此时，当前在 Raft 集群中的所有正常节点都将进入 Candidate 角色，并在一段随机的延时后向其他节点发出“给我投票”的请求，每个处于 Candidate 角色的节点都会把票投给第一个向自己发送“投票”请求的节点。若有 Candidate 节点在此轮投票中获得总节点数量一半以上的投票，则它会将自已标记为新的 Leader。否则重新进入下一轮投票，直到有节点获得半数以上的票数。每成功选举一次，新 Leader 的 Term（任期）值都会比之前 Leader 的增加 1。当集群中由于网络或其他原因的故障出现分裂又重新合并时，集群中可能会出现多于一个的 Leader 节点，此时，Term 值更高的一个节点将成为真正的 Leader。Raft 集群中的角色转换如图 2-2 所示。

除了服务的调度，SwarmKit 还提供了服务路由、负载均衡、故障处理和在线升级等能力。通常来说，在集群里创建的服务单元，其后端可以由多个运行在各个节点上的同种容器组成分担负载压力的逻辑单元，如图 2-3 所示。当有外部请求该服务时，SwarmKit 负责将这些请求均匀地分担到每个执行业务的容器里。当因部分容器意外崩溃而无法提供正常服务时，SwarmKit 会检测到这些问题，并自动地创建新的容器以确保每个服务后端的容器数量与预期保持一致。

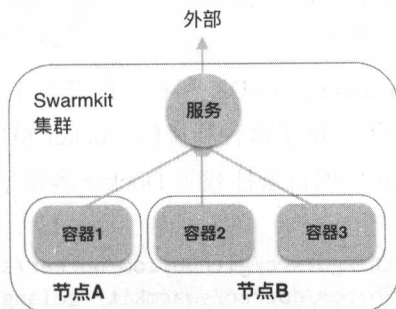


图 2-3 SwarmKit 集群中的服务

SwarmKit 依赖 Raft 协议保持集群状态的高可用，在 Raft 协议中，为了确保集群的状态和信息一致性，Leader 节点每次对存储的数据进行修改时，都需要告知所有 Follower 节点，并在获得半数以上 Follower 确认后，才能够将数据的修改持久化。随着节点数量的增加，确认消息的成本将成倍增长。因此，SwarmKit 又将所有节点划分成了 Manager 和 Worker 两种角色类型，这两类节点都会参与服务的调度，但只有 Manager 节点真正保存集群信息，并且参与 Raft 的选举过程。

这里的两种“角色”十分容易被混淆，它们在各自的术语中都被称为“Role”，因此需要通过上下文来识别。

- SwarmKit（以及 Swarm Mode）集群的角色：用来区分 Manager 或 Worker，它们的关键差异在于是否存储 Raft 数据以及是否接收用户的操作请求。
- Raft 集群的角色：指的是 Leader、Follower 或 Candidate，它们只存在于 SwarmKit 的 Manager 角色节点，各角色的关键差异在于是否主导 Raft 协议中的数据一致性协商。

如未特别说明，本书以下内容中提到的“角色”一般都是指 SwarmKit（以及 Swarm Mode）集群中的角色。

SwarmKit 节点作为 Manager 还是 Worker，是在节点进入集群时人为指定的，它同样可以在节点创建之后再行转换，只不过这种转换不会自动发生，需要人工指派。下个小节会介绍这个过程。

2.2.2 创建 SwarmKit 集群

SwarmKit 没有发布编译过的二进制版本，因为通常用户都会用 Docker 的 Swarm Mode 来间接地使用它。如果想直接使用 SwarmKit 工具，最直接的方式就是获取源代码进行编译。首先通过 Git 下载它的代码仓库，如下所示。

```
$ git clone https://github.com/docker/swarmkit.git
Cloning into 'swarmkit'...
... ..
```

编译 SwarmKit 需要一个 Golang 的 SDK 和相应的开发工具链，若手头上没有这样的环境，一种比较简便的办法是使用提供了这种环境的 Docker 镜像，例如官方的 `golang:1.8` 镜像。执行以下命令将 SwarmKit 代码目录挂载到 Docker 容器里进行构建。

```
$ docker run --rm -it \
    -v `pwd`/swarmkit:/go/src/github.com/docker/swarmkit \
    -w /go/src/github.com/docker/swarmkit/ golang:1.8 \
    make binaries
bin/swarmd
bin/swarmctl
bin/swarm-bench
bin/protoc-gen-gogoswarm
binaries
```

构建完成的文件存放在 SwarmKit 代码目录的“bin”文件夹内，将其中的 `swarmctl` 和 `swarmd` 两个文件拷贝到系统 PATH 变量指定的目录中，例如“`/usr/local/bin`”，如下所示。

```
$ sudo mv swarmkit/bin/swarmctl swarmkit/bin/swarmd /usr/local/bin/
```

在 SwarmKit 项目中，`swarmd` 是负责管理和维护集群的后台进程，`swarmctl` 是用户进行集群交互式操作的工具。不带任何参数的 `swarmd` 命令将用默认配置创建一个集群，并将当前节点作为集群的第一个 Manager 节点。默认配置会使用用户当前的目录存放 SwarmKit 运行过程中产生的各种数据文件，这并不是一种推荐的做法，下面这个命令会在系统后台启动 `swarmd` 进程，并指定集群的状态数据存放目录、监听的 Socket 文件位置、节点名字以及日志文件位置。

```
$ swarmd --state-dir /tmp/node-mgmt-01 --listen-control-api \
    /tmp/mgmt-01.sock --hostname mgmt-01 >/tmp/mgmt-01.log 2>&1 &
```

这样就得到了只有一个 Manager 节点的最小化 SwarmKit 集群。从严格意义来说，它还算不上一个真正的集群，但在这个单节点集群中已经可以执行 SwarmKit 的各种管理操

作，并且它也是创建更大规模集群的基础。

此时，使用 `swarmctl` 命令可以查看集群的信息。为了让 `swarmctl` 能够连接到 SwarmKit 集群，需要使用 `--socket` 参数或 `SWARM_SOCKET` 环境变量指定通信的 Socket 文件位置。`swarmctl node ls` 命令将列出集群节点的信息，如下所示。

```
$ export SWARM_SOCKET=/tmp/mgmt-01.sock
$ swarmctl node ls
```

ID	Name	Membership	Status	Availability	Manager Status
ea0b612...		ACCEPTED	READY	ACTIVE	REACHABLE *

为了向集群中添加更多的节点，还需要查询出当前 Manager 节点的 IP 地址和集群的 Join Token，如下所示。集群的 Join Token 相当于新节点加入集群的密码，只有提供了正确 Token 的请求才会被接受。此外，Join Token 还可被用来区分新节点角色是 Manager 还是 Worker。

```
$ ip addr show eth0
eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 ...
inet 172.31.27.16/20 brd 172.31.31.255 scope global eth0
... ..

$ swarmctl cluster inspect default
ID          : fe9vsdtfjco9vxjipwor1nj8x
Name        : default
Orchestration settings:
  Task history entries: 5
Dispatcher settings:
  Dispatcher heartbeat period: 5s
Certificate Authority settings:
  Certificate Validity Duration: 2160h0m0s
Join Tokens:
  Worker: SWMTKN-1-...-29kxz34bcz8gvzdwpimutxvp
  Manager: SWMTKN-1-...-34npzkvzoxss2s6tx7q1ss11s
```

注意 `swarmctl cluster inspect default` 这个命令的输出，它显示了名为“default (SwarmKit 的默认集群名字)”集群的两个“Join Token”。这两个 Token 分别对应了两种节点角色。

接下来向集群添加新的节点。将先前编译出来的 `swarmctl` 和 `swarmd` 文件拷贝到其他需要加入 SwarmKit 集群的节点上，同样放到 `PATH` 环境变量的目录里。这次在启动 `swarmd` 进程时，除了指定状态数据目录等信息，还需要提供 `--join-addr` 和 `--join-token` 参数表示加入已有集群（而不是新建集群），这里使用集群的 Worker Join

Token 将该节点作为 Worker 角色。

```
$ MANAGER_IP=<Manager 节点 IP>
$ JOIN_TOKENS=<Worker Join Token>
$ swarmd --state-dir /tmp/node-work-01 --hostname work-01\
  --join-addr ${MANAGER_IP}:4242 \
  --join-token ${JOIN_TOKENS} >/tmp/work-01.log 2>&1 &
```

重复该过程，添加更多节点到集群中，然后回到任意一个 Manager 节点上使用 `swarmctl node ls` 命令查看集群的节点信息，如下所示。

```
$ swarmctl node ls
```

ID	Name	Membership	Status	Availability	Manager Status
ea0b612...		ACCEPTED	READY	ACTIVE	REACHABLE *
lnxa1hn...		ACCEPTED	READY	ACTIVE	
bhk2f1r...		ACCEPTED	READY	ACTIVE	
...	...				

需要注意的是，所有 `swarmctl` 命令必须连接 Manager 节点的 Socket 文件才能使用，通常来说这也意味着只能在 Manager 节点上使用 `swarmctl` 命令。这是因为只有 Manager 节点才真正存储了集群的状态信息，而 Worker 节点仅仅用于执行服务容器。

使用 `swarmctl node promote` 和 `swarmctl node demote` 命令可以对节点的角色进行转换，它们的参数都是节点的 ID，前者将一个 Worker 节点提升为 Manager 节点，后者反之。

此外，SwarmKit 还为节点提供了“停机维护”的功能，命令是 `swarmctl node drain`，如下所示。

```
$ swarmctl node drain <节点 ID>
```

被指定的节点可用状态会被标记为“DRAIN”，此时如果该节点上运行有 SwarmKit 托管的容器服务，将被自动迁移到其他节点上运行，如下所示。

```
$ swarmctl node ls
```

ID	Name	Membership	Status	Availability	Manager Status
ea0b612...		ACCEPTED	READY	ACTIVE	REACHABLE *
lnxa1hn...		ACCEPTED	READY	ACTIVE	
bhk2f1r...		ACCEPTED	READY	DRAIN	

使用 `swarmctl node activate` 可将节点恢复为正常运行状态，如下所示。

```
$ swarmctl node activate <Node-ID>
```

2.2.3 在 SwarmKit 集群上运行服务

作为 Docker 的集群组件，SwarmKit 最重要的能力之一便是对服务的调度和管理。这些功能大多可以通过 `swarmctl service` 这个命令来操作。

创建服务的操作是 `swarmctl service create`，这个命令会通过 SwarmKit 的后台进程 API 在集群中创建使用指定镜像部署的服务，并根据当前集群各节点资源状态和服务的其他约束条件，将服务运行的 Task 调度到最合适的地方执行，如下所示。

```
$ swarmctl service create --name nginx --image nginx:1.11.1-alpine
t2neubvbyyqc8rynt09drjh19
```

在 SwarmKit 集群中，默认情况下所有节点都会参与服务的调度，包括 Manager 和 Worker 节点，这一点与过去的 Swarm 以及后面几章介绍的 Kubernetes、Mesos 等都不同。

使用 `swarmctl service ls` 命令可以查看当前集群运行的所有服务，如下所示。

```
$ swarmctl service ls
```

ID	Name	Image	Replicas
--	----	-----	-----
t2neubvbyyqc8rynt09drjh19	nginx	nginx:1.11.1-alpine	1/1

查看单个服务的详细信息可以使用 `swarmctl service inspect`，如下所示。

```
$ swarmctl service inspect nginx
```

```
ID           : t2neubvbyyqc8rynt09drjh19
Name          : nginx
Replicas      : 1/1
Template
  Container
    Image      : nginx:1.11.1-alpine
```

Task ID	Service	Slot	Image	Desired State ...	Node
-----	-----	----	-----	-----	----
rrk1vhg...	nginx	1	nginx:...	RUNNING	... mgmt-01

服务部署以后，还可以通过 `swarmctl service update` 命令来更新它的一些属性，比如容器副本的个数，如下所示。

```
$ swarmctl service update nginx --replicas 6
t2neubvbyyqc8rynt09drjh19
```

查看更新后的服务状况，可以发现名称是 `nginx` 的这个服务副本数变成了“6/6”，表示目标副本数量是 6，当前实际运行的副本数也是 6，如下所示。

```
$ swarmctl service ls
```

ID	Name	Image	Replicas
----	------	-------	----------

```
--
t2neubvbyyqc8rynt09drjh19  nginx  nginx:1.11.1-alpine  6/6
```

使用 `swarmctl service inspect nginx` 命令将列出其中每个容器副本所对应的 Task 信息，如下所示。

```
$ swarmctl service inspect nginx
... ..
Task ID      Service  Slot  Image      Desired State ...  Node
-----
rrk1vhg...   nginx    1     nginx:...  RUNNING             mgmt-01
fmfgcq...    nginx    2     nginx:...  RUNNING             mgmt-01
5f2vi8d...   nginx    3     nginx:...  RUNNING             work-01
w0s07ie...   nginx    4     nginx:...  RUNNING             work-01
qndf2cv...   nginx    5     nginx:...  RUNNING             work-02
xt2pm9j...   nginx    6     nginx:...  RUNNING             work-02
```

如果在其中一个节点上执行 `docker container ps` 命令，如下所示，会看到在该节点上运行的那些容器。

```
$ docker container ps
CONTAINER ID  IMAGE      COMMAND      ... ..  NAMES
1863bb173d97  nginx:...  "...        ... ..  nginx.1.rrk1vhg...
2ab1d45d0a3b  nginx:...  "...        ... ..  nginx.2.fmfgcq...
```

使用 `swarmctl service update -help` 可以看到在 SwarmKit 中允许动态更新的内容，其中 `--image` 这个参数经常被使用，它可以用来替换容器的镜像，这实际上可以用于升级服务的版本。例如下面这个命令可以将 `nginx` 服务的版本升级到 `1.11.3-alpine`。

```
$ swarmctl service update nginx --image nginx:1.11.3-alpine
t2neubvbyyqc8rynt09drjh19
```

执行完下面这个命令，服务的容器会被重启并替换成指定的新镜像。

```
$ swarmctl service ls
ID              Name  Image      Replicas
--
t2neubvbyyqc8rynt09drjh19  nginx  nginx:1.11.1-alpine  6/6
```

还可以加上更多的参数来控制服务升级的过程，例如下面这个命令会每隔四秒钟升级一部分服务，每次并行升级两个容器，直到所有容器都升级到新的镜像版本。

```
$ swarmctl service update nginx --image nginx:1.11.3-alpine \
--update-parallelism 2 --update-delay 4s
```

2.2.4 SwarmKit 集群的其他功能

除了已经介绍的节点和服务管理功能，SwarmKit 还提供了 Task、Secret 以及集群网络的管理。

```
$ swarmctl --help
... ..
Available Commands:
node          Node management
service       Service management
task          Task management
version       Print version number of swarm
network       Network management
cluster       Cluster management
secret        Secrets management
```

Task 是 SwarmKit 的最小管理单元，在当前版本里，一个 Task 实际上就对应一个容器。SwarmKit 抽象出 Task 的概念主要是为了未来能够让这个模型用在除容器外的集群资源管理，比如直接接管虚拟机，甚至是 Unikernel 的实例。Secret 是 SwarmKit 中对用户密钥信息的封装，这部分功能在讲解 Swarm Mode 的相应部分时再做介绍。

最后简单说一下 SwarmKit 中的 `network` 命令，它和 `docker network` 比较类似，不过在 SwarmKit 中只能看到后者的一部分网络。实际上，在这个命令中所能创建和管理的正是 Docker 网络中 `scope` 值为 `swarm` 的那些网络，它们通常都是建立在 SDN 之上的跨节点的虚拟网络，例如使用 `swarmctl network ls` 命令只显示驱动类型是 `overlay` 的那个网络，而不会看到本地节点上驱动类型为 `bridge` 的其他网络，如下所示。

```
$ swarmctl network ls
ID                               Name      Driver
--                               ----      -
xegwekbeisau53lmr6lyz2gak ingress overlay
```

2.3 Docker Swarm Mode

2.3.1 Swarm Mode 综述

Swarm Mode 指的是 Docker 整合了 SwarmKit 项目后增加的那部分功能，包括对容器

集群的管理、节点的管理、服务的管理和编排，以及其他的一些辅助功能。Docker 代码有许多地方直接 import 了“github.com/docker/swarmkit/”项目中的内容，也就是说 Docker 中与集群相关的功能实际上直接代理给了 SwarmKit 来实现。

从使用的角度上，Swarm Mode 在许多方面也都有着明显的 SwarmKit 影子。例如将集群节点分成 Manager 和 Worker，使用 Token 方式为集群添加节点；其中的许多概念，如 Service、Task、Secret 等，都直接沿用了 SwarmKit 中的相应称呼。同时，二者依然存在着一些差异，比如 Swarm Mode 弱化了 Task 的概念而增强了对服务编排的支持。

2.3.2 集群的创建与销毁

通过 Docker 的 `docker swarm` 命令集可以创建和管理集群，它的参数比 SwarmKit 中的 `swarmd` 命令还要简单。对于大多数情况，使用不带任何参数的 `docker swarm init` 命令就可以创建出一个新的集群，如下所示。

```
$ docker swarm init
Swarm initialized: current node (1tgyppr5dnz31ua18hxwft3up) is now a manager.
To add a worker to this swarm, run the following command:
    Dockerswarm join -tokenSWMTKN-1-42zcv0.....10ruju 172.31.31.164:2377
... ..
```

执行了创建集群的节点会自动成为该集群的第一个 Manager 节点，同时它打印了一个用来添加更多 Worker 节点的命令。这个命令的模式是 `docker swarm join-token <Token> <Manager-IP>`，其中的 `<Token>` 是一个用来区分请求加入者角色的序列码。要是忘记了，可以通过 `docker swarm join-token` 命令找回来（加上 `manager` 或 `worker` 表示要找回的 Token 种类），如下所示。

```
$ docker swarm join-token manager
To add a manager to this swarm, run the following command:
    docker swarm join --token SWMTKN-1-42zcv0.....48b01p 192.168.65.2:2377
$ docker swarm join-token worker
To add a worker to this swarm, run the following command:
    docker swarm join --token SWMTKN-1-42zcv0.....10ruju 192.168.65.2:2377
```

值得注意的是，添加 Worker 和 Manager 节点的 Token 值是不一样的。Token 不仅仅是用来保护集群不会随意混入无关节点的凭证，也是在节点加入时区分不同角色的依据。因此，作为集群的管理者，应该妥善保管 Token 序列的内容。特别是 Manager 角色的 Token，一旦泄露，则有可能使得入侵者能够随意向集群里加入新的管理节点，从而控制整个集群。

如果 Token 泄露了该怎么办呢？Docker 提供了一种 Token 轮换的机制，即在查看 Token 的命令中加上 `--rotate` 参数，如下。

```
$ docker swarm join-token --rotate manager
Successfully rotated manager join token.
To add a manager to this swarm, run the following command:
docker swarm join --token SWMTKN-1-42zcv0.....qdz76e 192.168.65.2:2377
```

每次执行这个命令都会改变相应角色加入集群的 Token 内容，并使得过去的 Token 失效，这只会影响加入的节点，已经在集群中的节点不受 Token 轮换的影响。将其他的节点加入集群的命令已经在显示 `join-token` 时完整地显示了，例如新增一个 Worker 节点只需 SSH 登录到目标节点，然后执行以下命令（注意其中的 Token 值和 IP 地址是变化的，请根据实际情况落实）。

```
$ docker swarm join --token SWMTKN-1-42zcv0.....10ruju 192.168.65.2:2377
This node joined a swarm as a worker.
```

仅需一条命令就完成子节点的加入，这对于使用者而言是十分友好的。

集群的安全隐患除了 Manager 节点的 Token 泄露，还出现在 Manager 节点本身——任何连接到 Manager 节点的用户都能直接操纵集群。为此 Docker 提供了一个给集群上锁的机制，它是由集群的一个 `autolock` 属性控制的，可以通过 `docker swarm update` 来修改，如下所示。

```
$ docker swarm update --autolock=true
Swarm updated.
To unlock a swarm manager after it restarts, run the `docker swarm unlock` command
and provide the following key:
```

```
SWMKEY-1-vhdf9LBAZ16qx8XoLH6TyPfQSaZjlaXWuyrG8aI3pH4
```

```
Please remember to store this key in a password manager, since without it you will
not be able to restart the manager.
```

这个命令会输出一个用于解锁集群的密钥值，请将其妥善保管。当 `autolock` 属性开启后，每当 Manager 节点的 Docker 服务被重启，就会进入锁定状态（本质上是锁了 Raft 数据文件），此时用户的所有与集群相关的操作都会被拒绝，如下所示。

```
$ sudo systemctl restart docker
$ docker node ls
Error response from daemon: Swarm is encrypted and needs to be unlocked before it
can be used. Please use "docker swarm unlock" to unlock it.
```

此时，如果需要通过这个节点对集群进行操作，需要在该节点执行一次 `docker swarm unlock` 命令，然后根据提示输入解锁的密钥，如下所示。

```
$ docker swarm unlock
Please enter unlock key: *****
```

如果忘记了密钥，只要当前集群中还有一个没有被锁定的节点，都可以在该节点上执行 `docker swarm unlock-key` 命令重新显示当前的解锁密钥。如果解锁的密钥意外泄露了，则可以通过 `docker swarm unlock-key -rotate` 命令更新解锁密钥的值。

节点解锁后，每次重启都会再次自动上锁。如果希望消除这个功能，只需取消集群的 `autolock` 属性，如下所示。

```
$ docker swarm update --autolock=false
Swarm updated.
```

如何让一个节点退出当前已经加入的集群呢？操作同样十分简单，只需 SSH 登录到节点上执行 `docker swarm leave` 命令，如下所示。

```
$ docker swarm leave
Node left the swarm.
```

此时会发生以下这几种情况。

- 如果这个节点是 **Worker** 节点，那么它将直接退出集群。Docker 集群会自动将该节点上的所有服务迁移到其他节点上继续运行。
- 如果这个节点是 **Manager** 节点，并且恰好是当前 **Manager** 中 **Raft** 集群的 **Leader**，则退出集群失败。如果确实要这么做，可以使用 `--force` 参数使其强制退出。
- 如果这个节点是 **Manager**，且当前集群中共有三个或以上 **Manager**，且该节点是 **Follower** 角色，那么它首先将自动降级为 **Worker**，使得 **Raft** 集群自动调整，然后再正常退出集群。
- 如果这个节点是 **Manager**，且当前集群中有且只有两个 **Manager** 节点，那么即使此节点是 **Follower** 角色，由于它若退出会使得 **Raft** 集群无法保持“半数以上成员”可用，因此退出集群失败。如果确实需要退出，同样可以使用 `--force` 参数强制执行。当集群中的所有节点都退出后，集群就被销毁了。

2.3.3 节点管理

在 Docker Swarm Mode 中管理节点的命令是 `docker node`。

当集群创建完成后，使用 `docker node ls` 命令可以看到各个节点的基本状态信息，如下所示。

```
$ docker node ls
ID                HOSTNAME      STATUS    AVAILABILITY  MANAGER STATUS
1h3s0f7tl... *   manager-1    Ready     Active         Leader
645i7ayla...     worker-1    Ready     Active
custji2pm...     worker-2    Ready     Active
```

以上显示的信息包括节点的 ID、主机名称以及状态。**STATUS** 和 **AVAILABILITY** 分别表示节点的健康性和可用性，正常情况下它们的值应该分别为 **Ready** 和 **Active**。**MANAGER STATUS** 用于区分节点的 Swarm Mode 角色（Manager 或 Worker）、Manager 的 Raft 角色（Leader 或 Follower）以及 Manager 节点的健康性。其中节点健康性状态的显示关系较复杂，如表 2-1 所示。

表 2-1 Swarm Mode 中的状态属性

节点角色和状态	STATUS	MANAGER STATUS
Worker 节点，正常	Ready	-
Worker 节点，故障	Down	-
Manager Leader 节点，正常	Ready	Leader
Manager Follower 节点，正常	Ready	Reachable
Manager Follower 节点，故障	Ready	Unreachable

也就是说，对于 Worker 节点，**MANAGER STATUS** 区域的值始终为空，而当节点有故障时，其 **STATUS** 属性会变为 **Down**。对于 Manager 节点来说，如果节点是 Leader 则显示为 **Leader**，Leader 节点一定不会有故障，否则会触发一轮 Raft 选举以重新产生新 Leader 将其取代。如果 Manager 节点是 Follower，则通过 **MANAGER STATUS** 的值区分其状态，而 **STATUS** 属性始终显示为 **Active**。

有两个命令可以用来获得与特定节点相关的信息，它们都接收一个节点 ID 作为参数。`docker node inspect` 命令获得的是节点的完整状态和属性，如下所示。

```
$ docker node inspect 1h3s0f7tl31jn0hqfsaeb5axr
[
  {
    "ID": "1h3s0f7tl31jn0hqfsaeb5axr",
    "Version": {
      "Index": 30
    },
    "Spec": {
```

```
"Role": "manager",
"Availability": "drain"
},
... ..
```

`docker node ps` 命令获得的是指定节点上运行的 Task 状态，如下所示。关于“Task”的概念会在下一个小节中介绍。

```
$ docker node ps 1h3s0f7t131jn0hqfsaeb5axr
ID          NAME      IMAGE      NODE      DESIRED CURRENT ...
vz45...    nginx.1   nginx:latest manager-1  Running  Running ...
```

与 SwarmKit 一样，Swarm Mode 中节点的角色是可以转换的。进行角色转换操作的命令是 `docker node promote` 和 `docker node demote`，它们的参数是要被转换的节点 ID。

转换 Worker 成 Manager:

```
$ docker node promote 645i7aylaci680a0skaelzgy
Node 645i7aylaci680a0skaelzgy promoted to a manager in the swarm.
```

转换 Manager 成 Worker:

```
$ docker node demote 1h3s0f7t131jn0hqfsaeb5axr
Manager 1h3s0f7t131jn0hqfsaeb5axr demoted in the swarm.
```

需要指出的是，所有与集群节点、服务以及密文的管理相关的操作都只能在 Manager 节点上进行。因此 Worker 节点是不能自己将自己提拔成 Manager 的。

对节点角色的更改实际上是更改节点属性的一种特例，对于更通用的情况，可以使用 `docker node update` 命令。比如 `docker node promote <node-id>` 和 `docker node demote <node-id>` 实际上等效于 `docker node update <node-id> --role manager` 和 `docker node update <node-id> --role worker`。

另一个比较常用的节点属性是节点可用性状态，它的值可以是 **Active**、**Pause** 或 **Drain**。其中 **Active** 状态表示节点可以正常使用。**Pause** 状态的节点不会再参与新的 Task 调度，但已经调度到该节点的 Task 可以继续运行。**Drain** 状态通常用于对节点进行维护或升级，处于此状态的节点将清理掉调度到当前的所有 Task 容器（Swarm Mode 会自动将这些容器在其他节点启动），同时也不再参与新 Task 调度。

可以使用如下命令更改节点可用性状态。

```
$ docker node update <node-id> --availability <active|pause|drain>
```

在前一小节中介绍到，为了让节点正常退出集群，需要登录到目标节点上，然后执行 `docker swarm leave` 命令。然而有时集群中的节点会发生故障，此时用户无法登录到节

点上将该节点退出，因此该节点的可用性状态始终显示为 Down，如下所示。

```
$ docker node ls
ID                HOSTNAME    STATUS    AVAILABILITY    MANAGER STATUS
1h3s0f7tl... *   manager-1   Ready     Active           Leader
645i7ayla...     worker-1    Ready     Active
custji2pm...     worker-2    Down      Active
```

针对这种情况，Swarm Mode 提供了一个 `docker node rm` 命令，这个命令只能作用在已经处于 Down 状态的节点，如下所示。

```
$ docker node rm custji2pmx9u6z7q8tigr0fx3
custji2pmx9u6z7q8tigr0fx3

$ docker node ls
ID                HOSTNAME    STATUS    AVAILABILITY    MANAGER STATUS
1h3s0f7tl... *   manager-1   Ready     Active           Leader
645i7ayla...     worker-1    Ready     Active
```

当被指定的节点当前处于 Active 状态时，执行 `docker node rm` 命令会失败，此时如果要移除该节点，需根据节点角色区分情况，如下所示。

- 指定节点是 Worker 角色：在这种情况下，如果需要移除指定节点，应该登录到节点上执行 `docker swarm leave` 操作。
- 指定节点是 Manager 角色：在这种情况下，首先需要将指定节点转换成 Worker 节点，然后根据转换后的节点状态决定重新执行 `docker node rm` 或 `docker swarm leave` 操作。

2.3.4 服务管理

在开始深入 Swarm Mode 的服务相关操作之前，有必要了解一下它的服务管理模型，如图 2-4 所示。

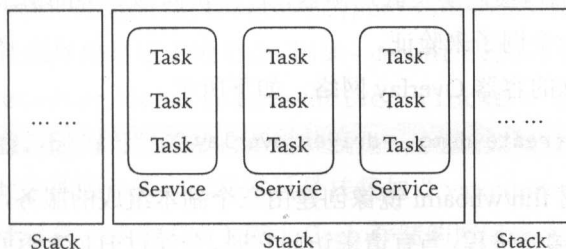


图 2-4 Swarm Mode 中的服务管理模型

在介绍 SwarmKit 时，已经简单地提到了 Task 和 Service 的概念。Task 是对底层运行单元的抽象，在当前的 Swarm Mode 中，一个 Task 实际上就是一个容器，它是 Swarm Mode 的最小调度单元，用于运行一个 Service 的容器副本。在 Swarm Mode 中没有专门管理 Task 的命令，不过通常直接使用 `docker container` 命令组来操作容器。

Service 是管理部署的单元，每个 Service 中可以包含一个或多个相同的容器副本，分布在集群的不同节点上，根据负载需要进行扩缩。此外，Service 还封装了负载均衡的功能，用于为属于同一个 Service 的分散容器提供统一的访问入口。本书中为了使行文通顺，在许多地方使用中文的“服务”替代“Service”一词，可以根据上下文判断指的是 Swarm Mode 模型的“Service”，还是泛指普通的服务。

Stack 是 Swarm Mode 与服务编排相关的概念，它将多个不同的固定 Service 关联在一起，进行整体的部署和升级，对外体现为由多种容器组合而成的复杂系统。

在 Swarm Mode 集群中，创建服务与在单个节点上创建容器颇有几分相似，只不过使用的不是 `docker container run`，而是 `docker service create`，如下所示。

```
$ docker service create --detach \
  --name web \
  --publish 8080:80 \
  --replicas=3 \
  nginx:1.11.1-alpine
```

这个命令创建了一个使用 `nginx:1.11.1-alpine` 镜像部署的服务，将其命名为“web”。它包含三个容器，并将容器中的 80 端口映射到外部的 8080 端口。集群里有那么多节点，这个服务映射的是哪个节点 IP 的 8080 端口呢？答案是所有的节点。现在访问集群中的任意一个节点 IP 地址的 8080 端口，都将看到一个 Nginx 的默认首页页面。

此外，Docker Swarm Mode 还悄悄地做了一些额外的配置工作。首先是将服务注册到了集群内的 DNS，这样与该容器处于同一网络的其他容器就能直接通过服务名称来访问此服务，这里的网络指的是由 `docker network` 命令管理的容器网络。然后为服务添加了一个由 Linux IPVS 管理的内核四层负载均衡器，任何访问该服务的请求都会由各个后端容器轮询处理。下面通过一个例子来验证。

首先创建一个独立的容器 Overlay 网络，如下所示。

```
$ docker network create demo --driver overlay
```

在这个网络中使用 `flin/whoami` 镜像创建由三个副本组成的服务，如下所示，这个镜像会运行一个监听 8000 端口进程，当有请求访问它时，会通过 HTTP 返回当前运行容器的 ID，据此可以判断出请求是哪个容器响应的。

```
$ docker service create --detach \
  --name whoami --replicas 3 \
  --network demo --publish 8000:8000 flin/whoami
```

创建一个用来访问被测试服务的容器，它需要与前者在同一个网络中，如下所示。

```
$ docker service create --detach \
  --name curl --network demo flin/curl
```

接下来需要进入 curl 服务的容器，它只有一个容器副本，使用 `docker service ps` 命令可以看到这个副本当前运行在哪个节点上，如下所示。

```
$ docker service ps curl
ID            NAME      IMAGE      NODE      DESIRED  CURRENT  ...
6czgya1...   curl.1    flin/curl  worker-1  Running  Running  ...
```

SSH 进入这个节点，使用 `docker container ps` 命令找到容器的实际 ID，然后从容器内访问 whoami 服务，如下所示。

```
$ docker ps | grep curl | awk '{print $1}'
ebed7646cf85
$ docker exec ebed7646cf85 curl -s whoami:8000
I'm d819ec8ff4cd
$ docker exec ebed7646cf85 curl -s whoami:8000
I'm c2bf30b3fb17
$ docker exec ebed7646cf85 curl -s whoami:8000
I'm a2ccf532b47d
```

可以看到，在容器中通过“whoami”这个名称可以直接访问相应的服务，并且多次访问此服务时，后端的各个 Task 容器是轮流处理请求的。

Swarm Mode 创建服务时有许多可用的参数，比较常用的有 `--port`、`--mount`、`--mode`、`--restart-condition` 和 `--constraint`。

- `--port` 是暴露服务端口的通用写法，例如前面 Nginx 服务例子的 `--publish 8080:80` 等同于 `--port mode=ingress,target=80,published=8080,protocol=tcp`。
- `--mount` 用于挂载存储卷或本地目录，例如 `--mount type=volume,source=data_vol_01,target=/var/data,volume-driver=flocker`。在集群中挂载存储目录时需要特别注意，因为当服务被自动迁移到另一个节点时，挂载目录存储的内容可能会丢失，因此集群服务的容器通常只是挂载如“/var/run/docker.sock”“/sys”这种有特定目的系统目录，或是采用 NFS、Ceph 存储的网络磁盘分区（配合 Flocker 或 Convoy 等工具更佳）。

- `--mode` 参数的值可以是 `replicated` 或 `global`，默认值是 `replicated`，此时可以用 `--replicas` 参数设置副本的个数。若将服务的 `mode` 设置为 `global`，则该服务会自动在每个节点上都运行一个副本。这个特性特别适用于部署监控和基础设施类的服务，每个新节点进入集群后都将自动创建这些 `global` 类型服务的 Task 容器。
- `--restart-condition` 参数服务的容器停止后，Swarm Mode 需要判断是否将其自动重启，默认值是 `any`，表示总是重启。可以将它设置为 `on-failure`，只在容器中进程退出且返回值不为 0 的时候才重启。或设置为 `none`，表示不自动重启该服务。
- `--constraint` 参数用于限制服务的容器可以被调度的节点，它可以根据节点的 ID、主机名、角色或是标签进行选择性的调度，其中标签是最灵活的一种方式。例如下面的操作给 `worker-1` 节点添加了一个 `zone=cn` 标签，然后在创建容器时指定服务只调度到具有这个标签的节点。

```
$ docker node update worker-1 --label-add zone=cn
$ docker service create --detach \
  --constraint node.labels.zone==cn ...
```

使用 `docker service create --help` 命令可以查看创建服务时其他的可用参数列表。

Swarm Mode 管理服务的命令与管理节点命令十分相似，如下所示。例如 `docker service ls` 命令用于查看服务列表，`docker service inspect` 命令用于查看指定服务的详细信息，`docker service rm` 命令用于删除一个服务。`docker service update` 用于更新服务的属性。

```
$ docker service ls
ID                NAME    REPLICAS  IMAGE                COMMAND
7zvnvn0ymy5x     web     3/3       nginx:1.11.1-alpine
a80zs876w8ft     curl    1/1       flin/curl
e7rn44ba6let     whoami  3/3       flin/whoami

$ docker service inspect web
[
  {
    "ID": "7zvnvn0ymy5xvk859nmfv16zz",
    "Version": {
      "Index": 225
    },
    "Spec": {
      "Name": "web",
      "TaskTemplate": {
        "ContainerSpec": {
```

```

        "Image": "nginx:1.11.1-alpine"
    },
    ... ..

$ docker service rm web
web

$ docker service update web --publish-add 8000:80
web

```

此外还有几个服务管理特有的命令。前面已经使用过的 `docker service ps` 命令用于显示指定服务的 Task 信息。在 Swarm Mode 中没有提供专门管理 Task 的命令，因此想列出集群中所有的 Task 需要遍历每个服务。这里有个小技巧可以将 `docker service ls` 和 `docker service ps` 命令组合起来，一次性查看所有 Task 的状态信息，如下所示。

```

$ docker service ls -q | xargs -n1 docker service ps
ID           NAME      IMAGE      NODE      DESIRED  ...
4gs065b94... curl.4    flin/curl  manager-1 Running  ...
ID           NAME      IMAGE      NODE      DESIRED  ...
60l95vxus... whoami.1  flin/whoami manager-1 Running  ...
5cngl0elv... whoami.2  flin/whoami worker-1  Running  ...
esipzyazs... whoami.3  flin/whoami worker-2  Running  ...
ID           NAME      IMAGE      NODE      DESIRED  ...
a7tdpqmr5... web.1     nginx:...  manager-1 Running  ...
1dsofk5fs... web.2     nginx:...  worker-1  Running  ...
e3xra5oru... web.3     nginx:...  worker-2  Running  ...

```

`docker service logs` 命令可以使用服务命令直接查看服务各个容器的日志，这个命令可以避免用户在集群中反复查找和登录各个节点去查看容器日志，对于排查服务故障十分有用。`docker service scale` 命令用于修改服务的容器副本数目，它其实是 `docker service update` 命令其中一个功能的快捷表示方式。以下这两个命令是等效的。

```

$ docker service scale web=5
$ docker service update web --replicas 5

```

在服务管理中，还有一个必须考虑的事情——服务的版本升级。在容器化的体系中，升级一个服务实际上就是替换服务镜像。然而替换镜像意味着需要重启容器，但在实际生产环境的应用场景中，一个服务中的所有容器并不是随时想停就能停的。不过由于 Swarm Mode 为每个集群中创建的服务都提供了与容器副本相关联的负载均衡器，因此如果服务本身是无状态的，就可以使用一种被称为“滚动升级”的方式完成不离线版本变更。

服务的“滚动升级”指的是将集群中处于同一个负载均衡器背后的副本实例逐步地替

换成新的版本，并动态更新负载的流量，以确保在整个升级过程中对外的服务功能不停止。服务的镜像也是服务属性之一，因此可使用 `docker service update` 命令进行更改，而逐步替换功能在 Swarm Mode 中是通过在升级服务镜像的同时限制更新并发数量实现的。下面以升级 Web 服务的镜像版本为例。

```
$ docker service update web --image nginx:1.11.5-alpine \
--update-parallelism 1 --update-delay 3s
```

在升级过程中，可以从另一个控制台窗口观察服务的 Task 容器变化过程，每隔三秒替换一个容器，直到所有的版本升级完成，如下所示。

```
$ watch 'docker service ps web | grep Running'
```

严格来说 Swarm Mode 中的“滚动升级”支持并不是非常完整，因为它只能做正向的滚动，如果升级出现错误，只能通过以低版本镜像为目标的再一次“滚动升级”来完成降级。但这对于通常的自动化运维而言已经足够了。

最后顺带介绍一个比较有用的技巧。在 Docker 中有许多 ID，例如节点 ID、服务 ID、网络 ID，等等，许多命令都需要和这些长长的 ID 串打交道，看起来颇为啰唆。实际上，在不引起歧义的情况下，Docker 允许使用 ID 的开头任意一个字符来替代这个长 ID 串，如下所示。

```
$ docker service create --detach --name nginx nginx:latest
```

此时如果在所有的服务里，只有这一个服务的 ID 是“m”开头的，那么可以直接这样引用它，如下所示。

```
$ docker service logs m
```

此时的“m”就指代前面的那串长 ID：mfhocfzr8uk6uxyranoqolyg2。如果有两个服务都是字母“m”开头，则 Docker 会提示错误，如下所示。

```
$ docker service logs m
Error response from daemon: service m is ambiguous (2 matches found)
```

那么用户至少需要指定开头两个字母，比如“mf”，以唯一确定 ID，以此类推。熟练掌握这个技巧可以避免许多拷贝和复制 ID 串的麻烦。

2.3.5 服务编排

服务编排指的是将一组服务按照它们之间的关联进行统一管理，以快速构建基于容器

的复杂应用的一种方式。Compose 一直以来都是 Docker 的服务编排工具，它通过一个 YAML 配置文件来描述各个容器的关联，然后提供一组命令来以整体视角操作所有容器。

在介绍集群级别的服务编排之前，先看一下在单主机上进行服务编排的过程。新建一个目录，创建名称为“docker-compose.yml”的文件，内容如下所示。

```
version: "3"
services:
  redis:
    image: redis:3.2.5-alpine
    networks:
      - demo-net
  app:
    image: flin/page-hit-counter:v1
    ports:
      - 5000:5000
    depends_on:
      - redis
    networks:
      - demo-net
networks:
  demo-net:
    external: false
```

“docker-compose.yml”是 Compose 默认的编排规则描述文件名，上述 YAML 文件描述了一个由两个服务和一个网络组成的系统，其中 Redis 服务提供了数据的外部缓存功能，而 App 服务是一个访问计数器，每次收到用户请求时，它就会从外部缓存中获取当前的访问总计数，将计数值加 1，然后写回外部缓存。通过使用外部缓存保存计数结果，App 服务就实现了无状态化，因此可以使用多个负载均衡的副本来分担用户的访问请求。

使用 Compose 将这个文件描述的服务快速创建出来，如下所示。

```
$ docker-compose up -d
```

连续访问当前节点的 5000 端口，会看到不断递增的访问计数，如下所示。

```
$ curl localhost:5000
You have hit this page 1 times. - Edition v1
$ curl localhost:5000
You have hit this page 2 times. - Edition v1
... ..
```

使用 docker-compose 命令可以批量地管理这些服务，例如将服务的所有容器停止，如下所示。

```
$ docker-compose stop
```

快速删除整个服务，包括服务涉及的所有容器、网络 and 存储等资源，如下所示。

```
$ docker-compose down
```

`docker-compose` 命令行的其他子命令及其说明如表 2-2 所示，其中的许多子命令与 `Docker` 工具本身十分相似，只是将作用范围扩大到了编排文件所描述的整个服务组。

表 2-2 Docker Compose 的子命令及其说明

命 令	说 明
<code>docker-compose up</code>	依据编排文件的描述创建所需资源并启动容器
<code>docker-compose down</code>	删除编排文件描述的与所有服务相关的容器、网络、存储资源
<code>docker-compose create</code>	创建所有被编排服务所需的资源，但不启动容器
<code>docker-compose start</code>	启动编排文件中涉及的所有容器
<code>docker-compose stop</code>	停止编排文件中涉及的所有容器
<code>docker-compose restart</code>	重新启动编排文件中涉及的所有容器
<code>docker-compose kill</code>	强行停止编排文件中涉及的所有容器
<code>docker-compose rm</code>	删除编排文件创建出的所有已停止的容器
<code>docker-compose pause</code>	暂停编排文件中涉及的所有容器
<code>docker-compose unpause</code>	恢复运行编排文件中涉及的所有容器
<code>docker-compose images</code>	列出编排文件中所涉及的所有镜像
<code>docker-compose ps</code>	列出由编排文件创建的所有容器
<code>docker-compose logs</code>	查看编排文件创建的所有容器日志，用不同颜色区分服务
<code>docker-compose events</code>	监听编排文件所创建的容器产生的所有事件
<code>docker-compose exec</code>	在编排文件所创建的指定容器中执行命令
<code>docker-compose pull</code>	获取编排文件中的所有来自仓库的镜像
<code>docker-compose push</code>	将编排文件中的所有自定义构建镜像推送到仓库
<code>docker-compose build</code>	构建编排文件中描述的所有需要构建的镜像
<code>docker-compose run</code>	单独启动编排文件中的某一个服务，这个命令主要用于调试
<code>docker-compose bundle</code>	从编排文件生成用于在集群部署的 <code>DAB</code> 文件
<code>docker-compose config</code>	检查编排文件语法，若有错误则提示，若无误则打印文件内容
<code>docker-compose port</code>	查看编排文件创建的任意服务端口映射的外部端口号
<code>docker-compose scale</code>	修改指定服务的容器副本数量
<code>docker-compose top</code>	查看编排文件所创建服务的 <code>PID</code> 和启动命令等信息

值得注意的是，使用不带参数的 `docker-compose up` 命令启动服务后，所启动的所有容器会保持在前台，并实时打印运行日志到控制台，这个行为与 `Docker` 命令行工具是一致的。但更多的时候用户希望 `docker-compose` 命令在启动完服务之后就所有容器放到后台运行，此时需要加上 `-d` 参数，如之前的例子所示。

从这个命令列表不难发现，Docker Compose 的所有操作都是基于服务编排描述文件的内容执行的。下面简单介绍一下这个编排文件的语法结构。

`docker-compose` 文件的顶级属性只能是 `version`、`services`、`volumes`、`networks`、`configs`、`secrets` 等固定的几个（其中 `configs` 和 `secrets` 不能用于单机的服务编排）。每个顶级属性下面会配置相应的资源描述。

1. 版本

`version` 属性仅仅描述当前编排文件所用的语法版本，Compose 的编排语法从诞生以来进行了几次大的版本升级，在本书截稿时（2017 年 10 月）的最新稳定版本是 3.3。读者可根据实际情况指定。

2. 服务

`services` 属性是整个编排描述中最核心的部分，包含了所需要运行的镜像信息、容器信息、副本数量以及对网络、磁盘、密文等资源的引用。

第二层级的属性是服务的名称，如下所示。

```
services:
  redis:
  ...
  app:
  ...
```

这表示这个服务编排组中一共包含两个服务，分别是 `redis` 和 `app`，每个名称后的部分则是对该服务的详细描述。

对于服务所用镜像的描述，可以使用 `build` 或 `image` 两种语法，前者表示构建需要从 Dockerfile 开始构建，此时进一步指定构建源所使用的工作目录和 Dockerfile 文件名称，如下所示。

```
services:
  webapp:
    build:
      context: ./dir
      dockerfile: Dockerfile-demo
```

后者表示从直接仓库获得指定镜像，如下所示。

```
services:
  webapp:
    image: redis
```


描述容器副本数量的属性是 `deploy.replicas`，同样在 `deploy` 属性下面的子属性还有滚动升级参数、重启动参数、资源配额限制、部署位置约束等。注意，`deploy` 属性对单机部署的服务编排无效，只能用于集群模式（即“应用栈”，见下一小节）。具体示例如下。

```
services:
  webapp:
    image: webapp
    deploy:
      replicas: 4                                #启动 4 个副本
      update_config:
        parallelism: 2                          #升级时每次替换 2 个容器，间隔 10s
        delay: 10s
      restart_policy:
        condition: on-failure                  #出错退出时自动重启，最多 3 次，间隔 3s
        delay: 5s
        max_attempts: 3
      resources:
        limits:
          cpus: '0.1'
          memory: 300M
        reservations:
          cpus: '0.01'
          memory: 100M
```

服务的容器可以使用端口、网络、存储、外置配置和密文等资源，并设置相关参数，但不包括对这些资源本身的描述，如下所示。

```
services:
  webapp:
    image: webapp
    volumes:
      - webapp_volumn:/opt/app/static          #挂载 webapp_volumn 存储卷
    ports:
      - "3000:udp"
      - "8000:8000"                          #指定端口映像
    networks:
      - webapp_network                       #加入 webapp_network 网络
    configs:
      - webapp_config                       #挂载 webapp_config 外置配置
    secrets:
      - webapp_secret                       #挂载 webapp_secret 密文
```

对资源的描述是写在各自单独的属性段里的。此外常用的属性还有指定容器启动顺序

关系的 `depends_on`、覆写容器入口命令的 `command`、配置环境变量的 `environment`、进行容器健康检查的 `healthcheck` 等，这里不再逐个展开介绍。

3. 存储卷

`volumes` 属性是对服务中所使用到的存储卷进行详细描述的地方，主要包含存储卷类型和参数。

下面就是一个使用了存储卷的服务示例。

```
services:
  db:
    image: postgres
    volumes:                #挂载 data 存储卷
      - data:/var/lib/postgresql/data
volumes:
  data:                    #名称是 data 的存储卷描述
    driver: local
    external: false
```

其中的 `external` 属性为 `false` 表示这个存储卷是由 Compose 管理的，会自动创建和删除，若用户希望自己预先创建存储卷，并自行管理存储卷生命周期，可将该属性设置为 `true`。

4. 网络

`networks` 属性是对服务中使用到的网络进行详细描述的地方，主要包含网络类型和参数。

下面就是一个指定了网络的服务示例。

```
services:
  app:
    build: ./app
    networks:
      - backend          #加入 backend 网络
networks:
  backend:              #名称是 backend 的网络描述
    driver: bridge
    external: false
```

网络的类型可以是 `bridge` 或 `overlay`，但 `overlay` 类型的网络只能用于集群模式。

在网络配置中同样有一个 `external` 属性，若为 `false` 表示这个网络是由 Compose 管理的，会自动创建和删除，若用户希望自己预先创建网络，并自行管理网络生命周期，可

将该属性设置为 `true`。

5. 外置配置

`configs` 属性是对服务中使用到的外置配置进行详细描述的地方，只对集群模式下的编排（即“应用栈”）有效，不能用于单机的服务编排。

下面就是一个指定了外置配置的服务示例。

```
services:
  app:
    build: ./app
    configs:
      - init_cf      #挂载 init_cf 外置配置
configs:
  init_cf:          #名称是 init_cf 的外置配置描述
    file: ./init.cf
    external: false
```

同样有 `external` 属性，表示外置配置的生命周期是否由 Compose 统一管理。

6. 密文

`secrets` 属性是对服务中使用到的密文进行详细描述的地方，只对集群模式下的编排（即“应用栈”）有效，不能用于单机的服务编排。

下面就是一个指定了密文的服务示例。

```
services:
  app:
    build: ./app
    secrets:
      - app_passwd   #挂载 app_passwd 密文
secrets:
  app_passwd:       #名称是 app_passwd 的密文描述
    file: ./secret_passwd
    external: false
```

完整的 compose 服务编排语法可参考官方文档^①。

Docker Compose 的编排文件名若由于特殊原因没有出现在当前目录，或没有命名为“`docker-compose.yml`”时，可以在执行相应命令之前使用“`-f <编排文件名>`”的方式指定所用编排文件的位置，如下所示。

① <https://docs.docker.com/compose/compose-file>


```
$ docker-compose -f path/to/my-compose-file.yml up
```

对于更复杂的场景，还可以将编排文件进行组合。例如将一个完整服务的非必须组件剥离到单独的编排描述文件，或是在不同环境运行时需要配置的不同部分写到单独的编排描述文件，然后在创建服务时通过多个 `-f` 参数依次指定这几个编排文件的位置，Docker Compose 会将它们中的内容合并为一个编排文件来执行，如下所示。

```
$ docker-compose -f docker-compose.yml -f docker-compose-dev.yml up
```

若 `compose-file.yml` 文件的内容发生了变化，例如替换了服务的镜像、修改了服务参数或是增加了新的服务，只需要在“`compose-file.yml`”文件所在的目录中再次执行 `docker-compose up-d` 命令，Docker Compose 会自动调整和重启相关的容器，使得运行的服务与描述保持一致，而无须手动重启或重建整个服务组。

2.3.6 应用栈的管理

描述服务编排的 Docker Compose 同样可以被用于在集群中部署服务和管理服务之间的关联，不过集群编排部署服务和单机的情况会稍有一些差异，主要体现在两者支持功能的差异上。譬如，在集群部署的服务通常会使用 Overlay 类型的网络，不支持 `--net=host` 模式的容器，不支持挂载本地设备（各节点设备可能不一致，存在隐患），但 Swarm 集群模式下的 Docker 增加了对外置配置和密文的支持。因此，虽然同样可以使用类似的 YAML 语法编排，可用的元素仍存在一些差异。

在集群中，每个服务在描述具体的编排细节时，有时会引用一些外部的文件，如外置配置或者密文的文件，在跨节点分发这些信息时，单独管理它们之间的关系会是一件麻烦事。Docker 曾经提出过一种实验性的集群部署专用打包文件来承载集群的服务编排配置，称为“分布式应用包”（Distributed Application Bundles，简称 DAB），保存为后缀名“`*.dab`”的文件。使用 Docker Compose 的 `docker-compose bundle` 命令可以将标准的“`docker-compose.yml`”文件转换成这种包。不过目前看来，这种打包格式实际很少有人使用，因此本书不打算详细介绍它。若读者对这部分内容有兴趣，请移步 Docker 文档。

在 Swarm 集群中，将通过编排方式部署到集群中的一组服务称为“应用栈（Stack）”。通过 Docker 命令行工具的 `docker stack` 下的子命令可以对集群中的应用栈进行管理。

首先将上个小节中的服务编排文件稍加修改，使它更适于在集群部署，如下所示。

```
version: "3.3"
services:
```

```
redis:
  image: redis:3.2.5-alpine
  networks:
    - demo-net
app:
  image: flin/page-hit-counter:v1
  ports:
    - 5000:5000
  depends_on:
    - redis
  networks:
    - demo-net
  deploy:
    replicas: 2          #增加多个副本
  networks:
    demo-net:
      driver: overlay    #更改网络驱动为 overlay
      external: false
```

使用 `docker swarm` 命令创建集群并添加几个节点。然后使用 `docker stack deploy` 命令创建部署到集群的应用栈，如下所示。注意 `docker stack` 命令只能用于已经开启 Swarm Mode 的节点。

```
$ docker stack deploy -c docker-compose.yml app
Creating network app_demo-net
Creating service app_app
Creating service app_redis
```

执行 `docker stack ls` 命令可以列出当前集群中的所有应用栈列表以及每个应用栈中包含的服务种类，如下所示。

```
$ docker stack ls
NAME      SERVICES
app       2
```

指定一个应用栈，用 `docker stack services` 命令列出该应用栈中的所有服务，如下所示。

```
$ docker stack services app
ID           NAME      MODE      REPLICAS IMAGE    PORTS
yrrdl...    app_redis replicated 1/1      redis:...
v19nq...    app_app   replicated 2/2      flin/p... *:5000->5000/tcp
```

使用 `docker stack ps` 命令可以直接看到应用栈里的所有容器，如下所示。

```
$ docker stack ps app
ID            NAME          IMAGE          NODE    DESIRED  CURRENT
svr6s...     app_redis.1   redis:...     node1   Running  Running
8pa51...     app_app.1     flin/page...  node2   Running  Running
pej7j...     app_app.2     flin/page...  node1   Running  Running
```

基于 Docker Service 的特性，一旦集群模式的容器指定了监听端口，它会占用整个集群中所有节点的指定端口。虽然只运行了两个容器的副本，但在集群的任意一个节点上访问 5000 端口，都能够访问到这个服务，如下所示。

```
$ curl localhost:5000
You have hit this page 1 times. - Edition v1
```

换一个节点访问，如下所示，由于服务的状态是外置在 Redis 中的，请求的数量会被累计。

```
$ curl localhost:5000
You have hit this page 2 times. - Edition v1
... ..
```

服务升级时首先要修改 “docker-compose.yml” 文件，例如将 flin/page-hit-counter: v1 镜像替换为 flin/page-hit-counter:v2，然后还要直接执行 `docker stack deploy` 命令，如下所示。

```
$ docker stack deploy -c docker-compose.yml app
Updating service app_redis (id: yrrdl...)
Updating service app_app (id: v19nq...)
```

再次访问 5000 端口，如下所示，会发现服务返回内容有了变化（末尾的 Edition v2，这是在新版本镜像中修改的），同时保存在 Redis 内存中的访问计数并没有被清零。这说明在升级过程中，只有发生了更改的容器（本例中的 App）被重新创建了，而未发生变化的容器（本例中的 Redis）并不会被连带重启。

```
$ curl localhost:5000
You have hit this page 3 times. - Edition v2
```

最后，执行 `docker stack rm` 命令删除指定的应用栈及所有相关的资源，如下所示。

```
$ docker stack rm app
Removing service app_redis
Removing service app_app
Removing network app_demo-net
```

2.3.7 外置配置和密文管理

在许多企业中，一些与环境相关的信息往往是由专门的运维团队管理的，开发团队并不关心在线上使用的数据库地址、密码以及其他与程序逻辑无关的事情。此外，这些数据可能需要定期地变动，将它们放在服务的镜像中并不太合适。Swarm Mode 的外置配置和密文就是为了支持这种职责分离的场景而引入的。

这两个功能是 Swarm Mode 在 Docker 1.13 以后新增的特性，它允许用户将一些运行时的配置信息提前保存到集群中，在启动服务时再动态地加载到容器里。实际上，在 Kubernetes 中早就有 ConfigMap 和 Secret 这两类对象了，同样是为了将软件的开发与运行解耦。具体来说，用户预先保存到集群里的信息，有一些是不带有保密性质的，比如一段普通文本信息或是某个测试用的缓存地址，另一些可能包含重要的敏感信息，比如生产运行环境数据库的密码。可以使用外置配置来管理那些不需要加密存储的普通数据，而包含敏感信息的数据则应该采用密文来管理。

与外置配置相关的操作定义在 `docker config` 命令下，它遵循 Swarm Mode 命令的一般模式，由几个主要的子命令组成，如下所示。

- `docker config create` 命令能够创建需要存储在集群的配置文件。
- `docker config ls` 命令用于查看当前集群中所有保存过的外置配置列表。
- `docker config inspect` 命令可以查看指定配置文件的详细属性。
- `docker config rm` 命令用于删除指定外置配置。

下面就以存储一个配置文件内容到集群，并在服务启动后读取为例，介绍外置配置的运用场景。首先创建一个外置配置对象，如下所示。

```
$ cat <<EOF | docker config create demo-config -
{
  "name": "demo"
}
EOF
```

创建一个服务，使用 `--config` 参数将预先创建到集群中的外置配置挂载到容器，如下所示。

```
$ docker service create \
  --detach \
  --replicas 1 \
  --name nginx \
  --config demo-config \
  nginx:alpine
```


外置配置默认挂载的位置是容器的根目录，如下所示。

```
$ docker service ps nginx      # 找到运行的节点
$ docker ps                    # 找到容器的 ID
$ docker exec -it <容器 ID> cat /demo-config
{
  "name": "demo"
}
```

使用 `docker service update` 可以动态地修改挂载的外置配置，例如将它移除，如下所示。

```
$ docker service update --detach --config-rm demo-config nginx
$ docker exec -it <容器 ID> cat /demo-config
cat: can't open '/demo-config': No such file or directory
```

注意，这个操作实际上重新创建了新的容器，因此容器的 ID 会发生变化，同时隐含一个服务重启的操作。

在挂载外置配置时，也可以指定挂载的目标文件，让挂载后的文件名和被挂载的外置配置名称不一样。还可以在外挂配置对象的名称上增加版本标识，实现配置版本化管理，如下所示。

```
$ cat <<EOF | docker config create demo-config-v1 -
{
  "name": "demo",
  "version": "v1"
}
EOF

$ docker service create \
  --detach \
  --replicas 1 \
  --name nginx \
  --config src=demo-config-v1,target="/etc/demo-config" \
  nginx:alpine
```

更新服务配置的时候，可以同时指定移除和添加的外置配置，从而实现配置内容替换，如下所示。

```
$ cat <<EOF | docker config create demo-config-v2 -
{
  "name": "demo",
  "version": "v2"
```

```
}  
EOF  
  
$ docker service update  
  --detach \  
  --config-rm demo-config-v1 \  
  --config-add source=demo-config-v2,target=/etc/demo-config \  
  nginx
```

与密文相关的操作由 `docker secret` 命令定义，包含的子命令与外置配置基本相同，如下所示。

- `docker secret create` 命令能够创建需要存储在集群的密文数据。
- `docker secret ls` 命令用于查看当前集群中所有保存过的密文列表。
- `docker secret inspect` 命令可以查看指定密文数据的详细属性。
- `docker secret rm` 命令用于删除指定密文。

与外置配置相比，密文的主要区别在于，它的内容在 `Swarm` 中存储和在容器网络中传输时都是以加密的形式存在的，只在目标节点中被解密到内存中，然后从内存映射到容器里。而外置配置是以普通磁盘文件的形式挂载进容器的。由于存在着加解密的额外开销和运行时的额外内存占用，单个密文内容被限制在 `500KB` 内。

密文的管理方式与外置配置几乎相同，下面这个例子把存储的内容从配置文件换成一串密码字符。首先把密码内容使用管道发送给 `Docker`，创建一个名称为“`demo-password`”的密文对象。发送给 `Docker` 的内容在集群内部会被自动加密存储，如下所示。

```
$ echo "这是密码的内容" | docker secret create demo-password -
```

接下来创建一个服务，在创建时用 `--secret` 参数给服务添加一个密文对象，如下所示。

```
$ docker service create \  
  --detach \  
  --replicas 1 \  
  --name nginx \  
  --secret demo-password \  
  nginx:alpine
```

当这个服务的所有容器启动时，会自动在“`/run/secrets`”目录下挂载一个与密文对象同名的文件，例如“`/run/secrets/demo-password`”，其中的内容是解密后的原始密文，如下所示。

```
$ docker service ps nginx    # 找到运行的节点  
$ docker ps                 # 找到容器的 ID
```



```
$ docker exec -it <容器 ID> cat /run/secrets/demo-password
```

这是密码的内容

通过这种方式，服务中的进程就可以在实际运行的时刻，从特定的目录获取所需的密文。而这些密文的内容可以比较方便地由管理集群和运行环境人员提供和更换。

在 Docker 17.06 版本开始，密文的位置也是可以随意指定的，格式与外置配置相似，如下所示。

```
$ docker service create \
  --detach \
  --replicas 1 \
  --name nginx \
  --secret source=demo-password,target=/opt/passwd \
  nginx:alpine
```

同样可以对服务挂载的密文进行动态替换，如下所示。

```
$ echo "这是更新过的密钥" | docker secret create demo-password-v2 -

$ docker service update \
  --detach \
  --secret-rm demo-password \
  --secret-add source=demo-password-v2,target=/opt/passwd \
  nginx

$ docker exec <容器 ID> cat /opt/passwd
```

这是更新过的密钥

2.4 Swarm Mode 的图形界面

2.4.1 Swarm Mode UI 现状

到目前为止，本书介绍的操作都是在命令行中实现的。而在实际管理集群时，特别是在需要持续观察集群状态和快速获取集群全貌时，总是通过输入这么多命令来刷屏查看信息就显得比较笨拙。

在 Docker 的企业版本中包含了一款管理 Swarm 集群的图形界面工具，被称为 Universal Control Plane（简称 UCP），它提供了对集群、节点、网络、存储、服务、应用组等全方位的可视化管理能力，如图 2-5 所示。

开源社区里也曾一度出现过许多 Docker 容器管理界面，但随着 Docker 增加了 Swarm 的组件，随后又从经典 Swarm 集群转移到 Swarm Mode 集群，许多相关的项目已不再继续更新了。在许多国内开发者参与贡献的项目中，相对知名的有数人云公司设计的 Crane 界面^①和个人开发者 Helyho 设计的 DockerFly 界面^②，这两个项目都对最新的 Swarm Mode 提供了支持，虽然代码活跃度都已不高，有兴趣的读者不妨保持关注。

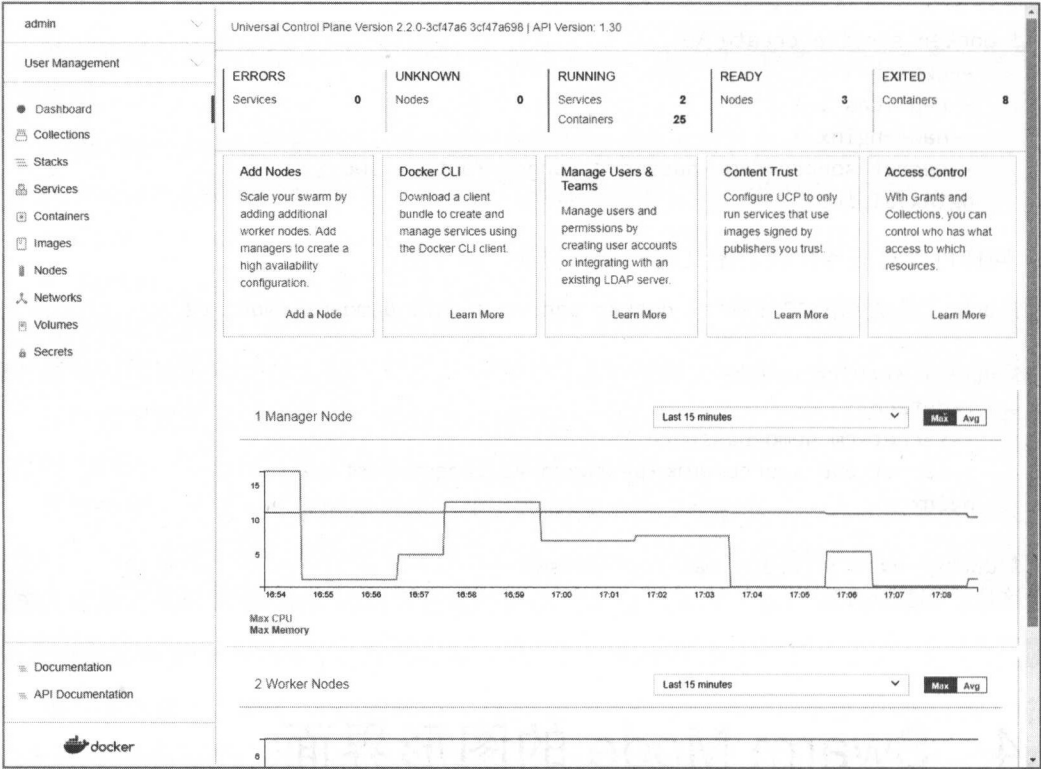


图 2-5 Universal Control Plane 的仪表盘页面

在社区项目中，最值得一提的是 Portainer^③，这个项目从 Docker 诞生后不久（2013 年 6 月）开始启动，一路跟随着 Docker 踏过单机时代的容器和经典 Swarm 时代的容器集群，现在完美支持了新的 Swarm Mode 集群，并保持着良好的活跃度和发布频率，是 Swarm Mode 集群可视化产品中值得一试的开源方案。

① <https://github.com/Dataman-Cloud/crane>
② <https://github.com/helyho/DockerFly>
③ <https://github.com/portainer/portainer>

2.4.2 Portainer

Portainer 的前端界面采用 Angular 框架, 代码结构和用户体验都十分优雅。由于 Portainer 采用本地文件（而不是数据库）的方式存储服务状态, 它在部署的时候没有什么额外的依赖, 在集群的 Manager 节点启动 Portainer 容器时即可使用, 如下所示。

```
$ docker run -d \  
  --name portainer \  
  --restart always \  
  -p 9000:9000 \  
  -v /var/run/docker.sock:/var/run/docker.sock \  
  -v /opt/portainer:/data \  
  portainer/portainer
```

容器中的“/data”目录被用来保存 Portainer 的状态数据, 建议将它挂载到主机上。这种简便和灵活设计的代价使它无法简单地进行水平扩展, 但考虑到集群管理系统通常只有管理员使用, 并不会有高并发访问, 所以算不上是严重的设计问题。Portainer 的数据目录存储的主要是用户组和集群连接信息等少量数据, 界面上展示的数据都是实时从集群里获得的, 所有在界面上与集群相关的操作也都会透明地传递给集群执行。

也可以使用 Service 的方式部署, 如下所示, 注意加上部署节点的选择条件, 确保它只会被部署在 Manager 节点上, 否则将无法获得集群信息。当然, 除非使用网络存储来挂载 Portainer 的数据目录, 否则使用这种方式部署的服务同样是不能水平扩容的。

```
$ docker service create \  
  --detach \  
  --name portainer \  
  --publish 9000:9000 \  
  --constraint 'node.role == manager' \  
  --mount type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock \  
  --mount type=bind,src=/opt/portainer,dst=/data \  
  portainer/portainer
```

部署完成后, 通过浏览器访问运行节点的 9000 端口（如果采用 Service 方式部署则可以是集群任意节点的 9000 端口）。首次进入时会提示设置一个 8 位数以上的管理员密码, 设置完成后, 使用 admin 用户和刚刚重置过的密码登录。

此时 Portainer 会问用户要操作哪个容器集群, 如果刚才创建容器时就是在 Manager 节点创建的（或通过 Service 方式创建的）, 则可以直接选择“Manage the Docker instance where Portainer is running”, 表示通过当前服务所在的节点接入集群。然后就进入了 Portainer 的主

仪表盘页面，如图 2-6 所示。



图 2-6 Portainer 的主仪表盘页面

注意图 2-6 左侧的导航栏菜单，它将 Portainer 的主要功能至上而下地划分为三部分。

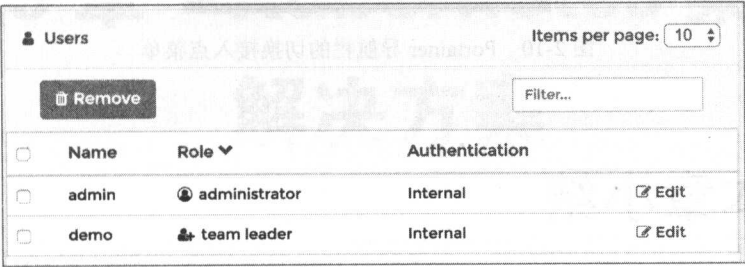
最上面的部分是“Active Endpoint”，这里可以用于切换管理的集群。Portainer 将每个被管理的集群称为一个“接入点（Endpoint）”，用户可以使用同一个 Portainer 界面管理多个集群，稍后会介绍接入点的添加和管理。

中间部分的“Enpoint Actions”是 Portainer 的主体功能，包含 Swarm 集群中的各种资源菜单。可以在不同的资源管理页面之间切换，所有的集群信息在这里都一目了然。此外，通过 Portainer 也能可视化地创建网络、存储、容器、服务等各种资源对象。

最底部的“Portainer Settings”部分只有管理员账号能看到，包含了用户管理（User management）、集群接入点管理（Enpoints）、仓库管理（Registies）和系统设置（Settings）这几个模块。在用户管理模块中，具有管理员权限的用户可以创建更多的普通用户账号并将这些账号划分成团队。接入点管理的模块用于添加更多 Portainer 可管理的集群，并将用户账号和可以访问的集群关联起来，当普通用户登录到 Portainer 时，只能看到和切换自己有权限的集群。仓库管理模块用于当用户使用 Docker Hub 的私有镜像组或是自建的私有仓库下载镜像时能够自动使用相应账号登录，解决通过 Portainer 部署容器时需要使用私有镜像的问题。

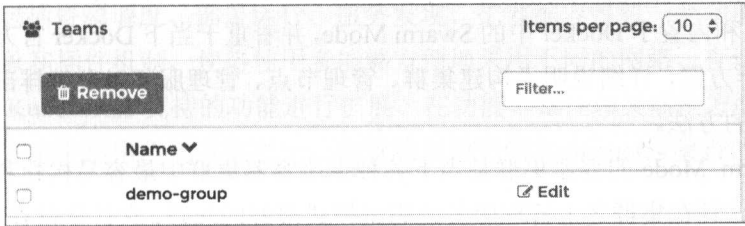
下面的例子展示了 Portainer 的多集群管理能力。首先进入左侧菜单的“Portainer Settings

→User Management” 页面，创建名为 “demo” 的用户，任意设置一个登录密码。然后进入 “Portainer Settings→User Management→Team” 页面，创建名为 “demo-group” 的团队，将 demo 用户设置为团队管理员（Team Leader）。此时用户页面的状态信息如图 2-7 所示，团队页面的状态信息如图 2-8 所示。



Users				Items per page: 10
<div>Remove</div> <div>Filter...</div>				
<input type="checkbox"/>	Name	Role	Authentication	
<input type="checkbox"/>	admin	administrator	Internal	<input checked="" type="checkbox"/> Edit
<input type="checkbox"/>	demo	team leader	Internal	<input checked="" type="checkbox"/> Edit

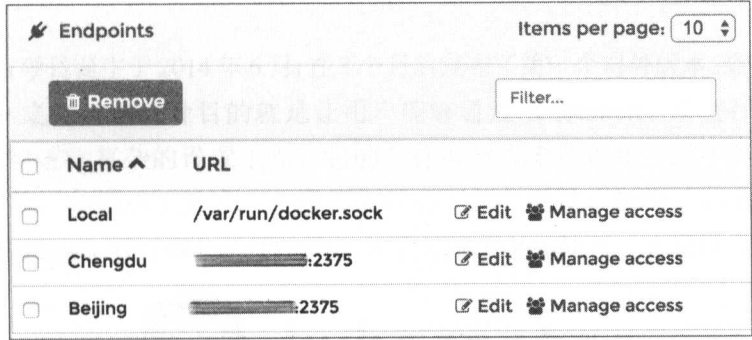
图 2-7 Portainer 的用户列表



Teams		Items per page: 10
<div>Remove</div> <div>Filter...</div>		
<input type="checkbox"/>	Name	
<input type="checkbox"/>	demo-group	<input checked="" type="checkbox"/> Edit

图 2-8 Portainer 的团队列表

接着进入 “Portainer Settings→Endpoints” 页面，添加要管理的其他集群地址，保存后单击各个接入点的 “Manage access” 按钮，赋予 demo-group 团队部分集群的管理权限，如图 2-9 所示。



Endpoints				Items per page: 10
<div>Remove</div> <div>Filter...</div>				
<input type="checkbox"/>	Name	URL		
<input type="checkbox"/>	Local	/var/run/docker.sock	<input checked="" type="checkbox"/> Edit	<input checked="" type="checkbox"/> Manage access
<input type="checkbox"/>	Chengdu	:2375	<input checked="" type="checkbox"/> Edit	<input checked="" type="checkbox"/> Manage access
<input type="checkbox"/>	Beijing	:2375	<input checked="" type="checkbox"/> Edit	<input checked="" type="checkbox"/> Manage access

图 2-9 Portainer 的接入点列表

然后重新使用 demo 用户登录，会发现点开左上角的接入点下拉菜单后，只能在有权

限的集群之间切换，如图 2-10 所示。

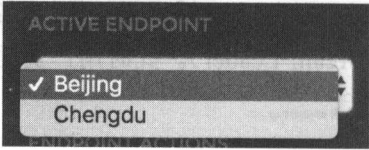


图 2-10 Portainer 导航栏的切换接入点菜单

2.5 本章小结

本章中从 Docker 的单机模式讲到集群模式的发展过程，介绍了经典 Swarm、独立的 SwarmKit 项目和内置于 Docker 中的 Swarm Mode，并着重于当下 Docker 官方主推的 Swarm Mode 容器集群方案，详细说明了构建集群、管理节点、管理服务以及编排部署由多容器构成的复杂服务的方法。

基于 Swarm Mode 的容器集群是当下各种主流容器集群中最容易构建和使用的一款，是理解和实践这种在集群之上管理服务的思维模式很好的起点。它让你站在指挥官的视角，以指挥整个军队而不是带领几个士兵的方式来看待你的集群。

第3章 Kubernetes 集群 解决方案

Kubernetes 是 Google 开源的容器集群管理系统。与 SwarmKit 不同，Kubernetes 并不依赖于 Docker 项目，而是被设计为一种通用的容器化应用管理方案（目前支持 Docker 和 Rkt）。它提供包括资源调度、部署运行、服务发现、扩容缩容等整套功能，通过与具体实现无关的抽象和插件机制，使得使用者能够方便地采用不同的网络、存储、容器等第三方组件，并对 Kubernetes 支持的功能进行扩展，在功能丰富性和灵活性方面是目前主流的几种容器集群方案中最出色的方案之一。本章将详细介绍 Kubernetes 集群的构建和使用。

3.1 Kubernetes 集群概述

3.1.1 Kubernetes 项目的起源

Kubernetes 项目诞生于 2014 年 6 月，在 3 个月后发布了第一个对外版本。最初的 Kubernetes 构建于 Docker 之上，其设计目的就是让用户能够通过 Kubernetes 来进行云端容器集群的管理，而无须进行复杂的设置工作。它的名称取自古希腊语中“舵手”的单词，Logo 是一个舵的形状，如图 3-1 所示。在项目公开后不久，微软、IBM、VMware、Docker、CoreOS 以及 SaltStack 等许多公司便纷纷加入了 Kubernetes 社区，为项目贡献源代码。



图 3-1 Kubernetes 的 Logo

由于 Kubernetes 出自 Google 之手，因此很容易让人想到 Google 的资源和管理系统：Borg。早在十几年前，Google 就已经部署 Borg 系统对来自于几千个应用程序所提交的任务进行编排、调度、启动、停止和监控等管理，以实现资源管理的自动化以及跨多个数据中心的资源利用率最大化。实际上，许多 Kubernetes 项目最初的开发者都曾工作在 Borg 系统上，这为该项目的架构质量和发展路线都提供了可靠的保障。Kubernetes 项目将 Borg 最精华的部分提取出来并加以改善，使普通的开发者也能够简单地部署和使用。

经过一年多的开发，Kubernetes 在 2015 年 7 月正式发布了 v1.0.0 版本，标志着其 API 和资源模型的成熟。在之后的几个版本里，Kubernetes 又逐步增加了 Deployment、DaemonSet、Job、CronJob、StatefulSet 等部署类型以及集群联邦等应用于大规模服务集群的功能。

值得一提的是，在国内的 Kubernetes 圈子里有不少优秀的活跃分子和用户。浙江大学 SEL 实验室、HyperHQ 公司和华为公司等高校和企业都是 Kubernetes 项目的重要代码贡献者。

3.1.2 Kubernetes 的结构

Kubernetes 的集群采用主从结构，所使用的服务器节点依据角色分为 Master 节点和 Node 节点，两种节点分别运行不同的服务进程。

图 3-2 展示了一个 Kubernetes 集群的部署结构，左侧的节点是一个 Master 节点。Master 节点的作用主要是控制和管理整个集群的状态并接收外部用户的操作请求。它主要运行三种服务进程，分别是 kube-apiserver、kube-scheduler 和 kube-controller-manager，具体作用如下所示。

- kube-apiserver

kube-apiserver 服务是整个 Kubernetes 集群管理系统的核心，也是部署 Kubernetes 系统时应该最先启动的组件。其他所有组件都会在启动时接入这个服务，以获取集群的信息或注册自己的信息。这些功能主要是通过管理 Etcd 存储的集群状态信息并对外提供 Restful API 实现的。

- kube-scheduler

kube-scheduler 服务是 Kubernetes 中负责用户服务调度的子模块，它能够根据集群当前的资源使用情况选择适合运行特定服务的节点。由于调度的策略与实际的底层硬件关系比较密切，Kuberentes 将这部分功能设计为单独组件，使其具有可替换性和可扩展性，从而能在未来适配更多现有的系统。

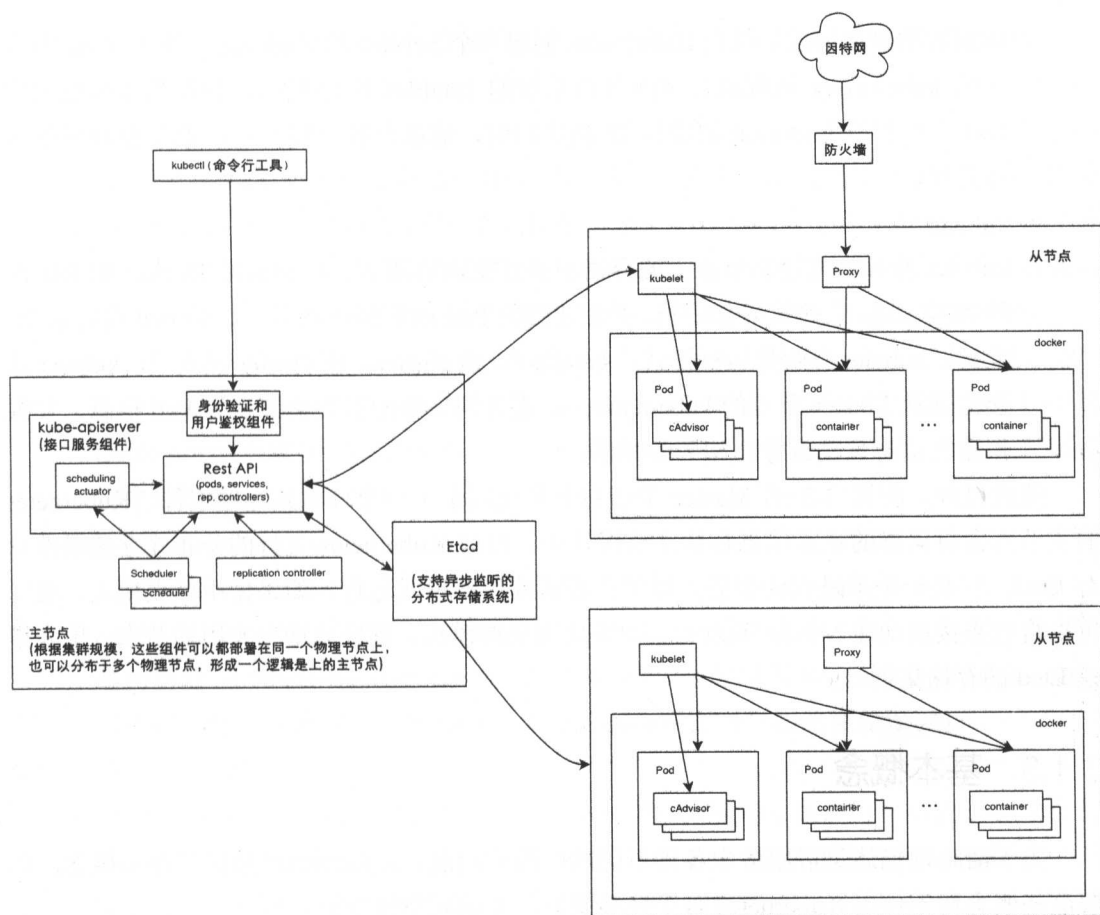


图 3-2 Kubernetes 的部署架构图

• kube-controller-manager

kube-controller-manager 服务负责管理 Kubernetes 系统中各种资源的状态。具体来说, 依据其监控的资源种类, 它包括管理 Pod 副本数量的 ReplicationController、管理 Service 映射的 EndpointController、管理多租户空间的 NamespaceController 等十多种不同种类的模块, 以 Goroutine 协程的形式运行。

除此之外, 主节点上还可能运行 kube-dns 和 kube-discovery 等可选的服务。

图 3-2 的右侧展示了两个 Node 节点, Node 节点是实际执行用户任务的地方。它们都需要运行两种 Kubernetes 服务进程, 分别是 kube-proxy 和 kubelet, 具体作用如下所示。

• kube-proxy

设计 kube-proxy 服务是为了解决从 Node 节点中的 Pod 对 Kubernetes 特定 Service

访问时的路由问题。每当 Kubernetes 创建一个 Service 的 Endpoint，各个 Node 节点上的 kube-proxy 进程就会修改节点系统的 Iptables 网络规则，使得当该节点上的 Pod 访问相应 Endpoint 的虚拟 IP 和端口时，请求会被分发到正确的节点和容器进行处理。

- kubelet

kubelet 服务是直接和节点上的容器服务打交道的通道，目前支持 Docker 和 Rkt 两种容器实现。它接收外部请求，并创建新的 Pod 或汇报所在节点上的 Pod 运行状态。此外，kubelet 有两种运行模式：standalone 或 cluster。在 cluster 模式下，kubelet 还会定期从 Master 节点的 kube-apiserver 服务同步分配到当前节点的 Pod 信息，并根据这些信息启动或停止相应的容器。

除此以外，在图 3-2 的 Master 节点右下角还展示了一个分布式存储组件，Kubernetes 的主节点会将集群的状态信息存储在该组件中。目前 Kubernetes 支持的分布式存储组件只有 Etcd，它是一个单进程的服务，以单点模式或集群模式运行。如果使用单点模式，通常可以将它直接启动在 Master 节点上。如果使用集群模式，则应该增加专用的节点，用来搭建 Etcd 的存储集群。

3.1.3 基本概念

为了清晰地描述集群服务中各种不同的资源和功能，Kubernetes 抽象了许多概念，理解这些概念对于使用 Kubernetes 是十分必要的，下面列举其中的一些。

- Pod

Pod 表示的是一个容器或多个容器的组合，它是 Kubernetes 最基本的调度和操作单元。在同一个 Pod 中的容器总会被调度和部署到同一个节点上，并共享相同的数据卷和网络栈。这意味着在 Pod 里定义的几个容器能够同时挂载同一个外部数据卷，还能通过 localhost 作为地址相互访问，这个特性对于部署关系紧密的服务具有十分重要的作用。

- Deployment

Deployment 是 Kubernetes 对于后台服务部署操作的抽象。每个 Deployment 对象有一个部署目标（通常是一系列 Pod 副本）来保存所有部署描述信息的历史。通过这些信息，Kubernetes 便能对目标的部署配置进行修改和回滚。

实际上，Deployment 所保存的部署描述信息是用来创建 ReplicaSet 对象的，每个

Deployment 对象都会对应一个运行着的 ReplicaSet 对象，后者实际管理 Pod 副本的运行。

- ReplicaSet

ReplicaSet 用于记录和控制 Pod 副本的数量，它使用预定义模板自动创建指定数量的 Pod 副本实例，并运行到不同的 Node 节点上。一旦指定了副本数量，Kubernetes 就能确保集群中始终有指定数量的 Pod 副本在运行，如果实际数量少于指定数量，Kubernetes 会启动新的 Pod，反之则会关闭多余的 Pod，以维持副本数量不变。

通常用户不会直接操作 ReplicaSet 对象，而是通过 Deployment 对象间接完成。例如在之后的章节里会看到，修改特定 Pod 运行副本数量，用户应该通过操作相应的 Deployment 对象来删除 Pod 所属的 ReplicaSet 对象，然后重新创建一个符合目标副本个数的 ReplicaSet 对象。

- ReplicationController

ReplicationController 是在 Kubernetes 早期版本中用于管理 Pod 副本对象的概念，从功能上来说，它同时包含了 ReplicaSet 和 Deployment 的作用，但不具有部署版本管理的功能。ReplicationController 在 1.2 版本以后，建议使用 ReplicaSet 和 Deployment 替代它，但它目前在一些 Kubernetes 的 Example 案例中依然被使用。

- Service

Service 是 Kubernetes 集群对外提供的用户业务功能抽象，表现为一个独立的虚拟 IP 和端口。它的后端实际是由单个 Pod 或许多 Pod 的副本组成的容器集合，由 Kubernetes 提供实际访问的路由能力。通过这层抽象，Kubernetes 能够在后端完成例如现象服务部署和切换、负载均衡、依据资源情况的节点调度等，而用户并不需要关心这些烦琐的工作是如何完成的。

- DaemonSet

DemonSet 的作用与 ReplicaSet 有些类似，但它并不是用于确保指定 Pod 副本在集群中的总数不变的，而是用于确保指定 Pod 在每个 Node 节点上都有且只有一个运行的副本。每当集群中新增一个 Node 节点时，在 Master 节点上的 DaemonSet 控制器就会通知该节点创建相应的 Pod 副本实例。

- StatefulSet

StatefulSet 在 Kubernetes 的 1.5 版本前称为 PetSet，是用于部署和运行有持久化状态服务的方式，它与 ReplicaSet 属于同一类型的概念，代表指定数量的 Pod 副本集合。由 StatefulSet 管理的每个 Pod 副本都有一个唯一的名称，并在集群的 DNS 中添加一

个单独的域名记录，在该副本的整个生命周期里，它所获得的域名和挂载的数据卷都会保存一致，即便进行升级替换了 Pod 的内容，原先存储的数据依然会跟随它。

- Job

Job 与 ReplicaSet 有几分相似，但它不像后者被用于运行长期存在的后台服务，Job 是用来批量执行一次性任务的。每个 Job 同样会对应一个或多个 Pod，这些 Pod 所执行的都是有限时间长度的进程，当进程结束后，依据进程的退出状态码，Job 将被标记为成功或失败。

- CronJob

CronJob 在 Kubernetes 的 1.4 版本前称为 ScheduledJob，在集群中的角色像是 Linux 系统的 cron 命令，它是简化 Kubernetes 集群管理十分有用的一种资源类型。每个 CronJob 除了包含指定的一次性任务，还有一个使用 Cron 表达式指定的执行时间，每当集群的系统时间符合该表达式时，相应的任务就会被执行。

- Ingress

Ingress 本质上是一个负载均衡服务，它被用来将集群中的用户服务通过统一的 IP 地址暴露给外界。它的实现可以基于多种不同的具体负载均衡工具，统一 IP 地址的好处在于，用户可以为该地址绑定域名，从而通过自定义的域名直接访问集群内部署的服务。

- ConfigMap

ConfigMap 是 Kubernetes 存储应用配置的对象。对于使用“Cloud Native 架构”的服务，将服务进程本身的代码与它的配置区分开管理，是一个非常好的实践。ConfigMap 使得用户能够在服务部署时动态地将配置与服务进行绑定，而不需要在代码中预先设置。

- Security

Security 是用于存储私密配置信息的对象。它的作用与 ConfigMap 相似，然而使用 Security 存储的配置信息在实际的存储服务（例如 Etcd）中是以密文方式保存的，它十分适合保存密码、密钥以及其他不应该被配置的使用者直接看到的数据。

- Namespace

Namespace 主要用于多租户的隔离，Kubernetes 中除了 Node 和 Label 以外的大多资源，例如 Pod、Service、ResourceQuota、LimitRange 等，都是创建在特定 Namespace 里的（默认使用名为 default 的 Namespace）。在不同 Namespace 中的资源不能感知到对方的存在，也不能相互访问，这样相当于在集群内部又划分了子环境，有利于

区分软件的测试和产品运行环境，同时过滤了查询结果中出现不必要的项目，减少了大规模集群管理难度。

- Node

Node 是 Kubernetes 中对集群中的 Node 节点资源的抽象，每个 Node 对象对应了实际的服务器节点。当有新的 Node 节点加入集群时，Kubernetes 就会自动为它创建一个 Node 对象。通过操作这些对象，使用者能够方便地对每个节点进行管理，例如添加 Label、定制部署规则、快速 SSH 登录、临时下线维护等。

- Label

Label 是用于标记 Pod、Node、Namespace 或其他任何资源对象的键值对，主要用于在查询或选择时对资源进行条件过滤。Label 机制也是 Kubernetes 中组织 ReplicaSet、DaemonSet、Job 和 Service 等概念的基础，它被用于在所有 Pod 对象中识别出哪些 Pod 属于同一组 ReplicaSet 或 DaemonSet 以及哪些 Pod 提供了同一个 Service 对象的服务。

- Annotations

Annotations 同样是附属于其他资源上的一种键值对标记，但这些标记通常被作为 Kubernetes 的系统服务创建资源时的特殊参数。用户无法使用它在通过 API 查找资源时进行过滤和匹配。每种资源上能够标记的 Annotations 键的名称是预定义好的，而且每个键都具有特定的作用。

- ResourceQuote

ResourceQuote 使得用户能够以 Namespace 为单位给每个租户设置资源用量的限制，这个特性体现了租户的意义：避免单个用户耗尽所有服务资源以及作为计费功能的基础。Kubernetes 的 ResourceQuote 提供两种类型的资源限制：硬件资源用量（CPU 和内存）以及可创建对象总数（Pod/Service/ConfigMap/……）。

- LimitRange

LimitRange 提供了 Kubernetes 中的另一种限制方式。相比于 ResourceQuote 对每个 Namespace 资源总量的限制，LimitRange 限制的是每个 Pod 或容器能够使用的 CPU 和内存资源量。此外，LimitRange 也可以被用来配置每个 Namespace 创建 Pod 时被默认赋予的可用资源值。

- VolumeClaim

VolumeClaim 是 Kubernetes 中用于声明可用数据卷的对象，这样做是为了将数据卷的资源分配和使用数据卷的服务部署解耦。每个 VolumeClaim 对象代表了一定容量

的存储资源，并使用元数据对其加以描述，要想使用存储资源的服务只需声明所需的容量和其他特征，由 Kubernetes 自动进行关联管理。

前文介绍了这么多概念，如果不结合实际的例子确实很难理解它们的作用，这也是 Kubernetes 的体系学习曲线相对陡峭的原因。然而一旦你熟悉了这些概念，就会发现 Kubernetes 在模型设计上的精妙之处：几乎所有的资源都能够使用相同的命令和 Yaml 模板文件进行管理。

在本书之后的章节中，还会用到这些概念，并对与它们相关的操作和使用进行更详细的介绍。

3.2 部署 Kubernetes 集群

3.2.1 使用 Minikube

Minikube^①是 Kubernetes 官方维护的一个在虚拟机上快速搭建单节点 Kubernetes 的工具，原本是为了简化本地开发和测试环境的搭建，支持 VirtualBox、WMWareFusion、KVM 和 Xhyve 的虚拟机。

和 Kubernetes 的其他组件相似，Minikube 本身是用 Go 语言开发的单个二进制文件程序，下载到系统的 PATH 环境变量中的目录即可使用。以 Linux 系统为例，安装 Minikube 工具的命令如下所示。

```
$ curl -Lo minikube \
https://storage.googleapis.com/minikube/releases/v0.22.1/minikube-linux-amd64
$ chmod +x minikube
$ sudo mv minikube /usr/local/bin/
```

Minikube 默认使用 VirtualBox 创建 Kubernetes 集群，因此在使用之前应该确认系统中已经安装当前最新版本的 VirtualBox，并且启动后台服务，然后只需输入以下命令。

```
$ minikube start
```

屏幕提示“Kubectl is now configured to use the cluster”说明表面 Kubernetes 的集群已经启动完成了。Kubectl 是 Kubernetes 中用于控制集群的工具，可以通过以下命令安装。

^① <https://github.com/kubernetes/minikube>

```
$ curl -Lo kubect1 \
http://storage.googleapis.com/kubernetes-release/release/v1.7.6/bin/linux/amd64/
kubect1
$ chmod +x kubect1
$ sudo mv kubect1 /usr/local/bin/
```

在 Minikube 部署完 Kubernetes 集群时，已经顺便创建了 kubect1 工具的配置。因此无须更多的操作就可以直接使用 kubect1 命令与刚刚部署的集群进行交互了，例如查看集群的节点信息，如下所示。

```
$ kubect1 get node
```

此外，使用 minikube dashboard 命令将显示 Kubernetes 的 Dashboard 控制台界面的地址。当暂时不再使用 Kuberentes 集群时，可以用 minikube stop 命令将它的虚拟机关闭。而 minikube delete 命令将彻底销毁整个集群。

输入下面这个命令可以查看 Minikube 的介绍和参数说明。

```
$ minikube --help
```

3.2.2 使用 kubeadm

kubeadm 是 Kubernetes 1.4 以后版本推荐的正式环境部署方式，也是一个 Go 语言编写的单文件二进制程序，替代了过去的 kube-up.sh 部署脚本。

首先，kubeadm 命令需要启动一个独立的 kubelet 服务，然后通过 kubelet 服务使用 Docker 来运行所有其他的 Kubernetes 组件，如图 3-3 所示。

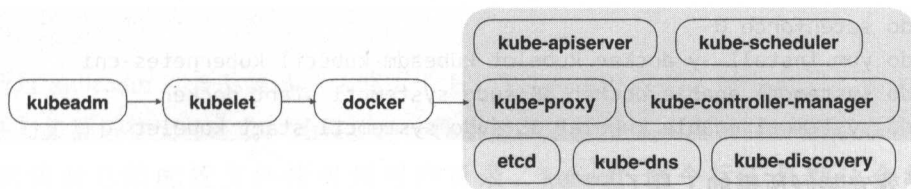


图 3-3 使用 kubeadm 工具创建 Kubernetes 集群

通过 kubeadm 工具进行 Kubernetes 部署前，首先需要安装相应的工具。这些工具除了 kubeadm 本身，还包括 kubelet 和 Docker。此外，kubeadm 只是负责将集群构建起来，不能和构建完的 Kubernetes 集群交互，因此还需要 kubect1 工具。

值得庆幸的是，这些工具的安装在主流的 Linux 操作系统中都十分简单。

1. 在 Ubuntu 16.04 系统中安装 kubernetes 工具

Ubuntu 系统使用 Debian 系列的 Apt 工具管理软件包，在系统的“/etc/apt/sources.list.d/”目录下增加一个包的源仓库文件，命名为“kubernetes.list”，写入一行内容，如下所示。

```
deb http://apt.kubernetes.io/ kubernetes-xenial main
```

保存文件，执行以下命令即可完成所有必需工具的安装工作。

```
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -  
$ sudo apt-get update  
$ sudo apt-get install -y docker.io kubelet kubeadm kubectl kubernetes-cni
```

2. 在 CentOS 7 系统中安装 kubernetes 工具

在 CentOS 中安装这些工具的方式与 Ubuntu 相似，不同之处在于，CentOS 系统默认采用 Yum 作为包管理工具。首先同样需要添加源仓库，在“/etc/yum.repos.d/”目录下新建名为“kubernetes.repo”的文件，写入以下内容。

```
[kubernetes]  
name=Kubernetes  
baseurl=http://yum.kubernetes.io/repos/kubernetes-el7-x86_64  
enabled=1  
gpgcheck=1  
repo_gpgcheck=1  
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg  
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
```

然后执行安装命令（需要关闭 SELinux），如下所示。

```
$ sudo setenforce 0  
$ sudo yum install -y docker kubelet kubeadm kubectl kubernetes-cni  
$ sudo systemctl enable docker && sudo systemctl start docker  
$ sudo systemctl enable kubelet && sudo systemctl start kubelet
```

这样就安装好所需的工具了。

最后不论是 Ubuntu 还是 CentOS 的用户，在默认情况下，新安装的 Docker 工具只有 root 用户能直接执行，普通用户每次都需要使用 sudo 来获取执行权限，这样十分不便。可以通过将当前用户添加到名称为“docker”的用户组来解决这个问题，如下所示。

```
$ sudo usermod -aG docker `whoami`
```

添加完成后退出当前用户，重新登录就可以直接执行 docker 命令了。

使用 `kubeadm` 部署集群的过程很简单，先选择一个作为 Master 节点的服务器，执行 `kubeadm init` 命令，如下所示。

```
$ sudo kubeadm init
... ..
Kubernetes master initialised successfully!
You can now join any number of machines by running the following on each node:
kubeadm join --token 90317f.4787167787e64c48 172.31.31.182
```

这个命令默认将部署当前最新版本的 Kubernetes 服务，可以使用 `--use-kubernetes-version` 参数指定需要部署的版本，例如 `sudo kubeadm init --use-kubernetes-version v1.7.6`。

如果部署没有问题，在输出的最后一行将给出用于部署 Node 节点的 `kubeadm join` 命令参数，请将该命令中的参数值记录下来。然后，在每个需要被作为 Node 节点的服务器上分别执行刚刚显示的这个命令，如下所示。

```
$ sudo kubeadm join --token <cluster-id> <master-ip>
... ..
Node join complete:
* Certificate signing request sent to master and response received.
* Kubelet informed of new secure connection details.
Run 'kubectl get nodes' on the master to see this machine join.
```

注意，在 Node 节点运行 `kubeadm join` 命令时需要使用 Root 用户(或使用 `sudo` 执行)，因为 `kube-proxy` 服务需要操作系统的 `iptables` 服务，必须获得管理员权限。继续在所有需要加入集群的节点上依次执行 `kubeadm join` 命令，一个简单的 Kubernetes 集群就搭建完成了。

通过 `kubeadm` 部署的集群默认监听了 HTTPS 的 API 接口，并且开启了调用者身份验证。在与集群进行交互之前，还需要做一件事情，就是将生成在 Master 节点系统目录下的包含密钥信息的配置文件拷贝到用户目录。`kubectl` 命令行工具默认会将用户的“.kube/config”文件作为配置文件使用，如下所示。

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

现在回到 Master 节点上，使用 `kubectl get nodes` 命令查看当前集群中的所有节点。若返回类似下面这样的节点列表信息，表明部署正常。


```
$ kubectl get nodes
NAME      STATUS   AGE
node1     Ready    3m
node2     Ready    1m
... ..
```

对于国内用户而言，通过 `kubeadm` 部署 Kubernetes 集群的过程可能不会太顺利。在 `kubeadm` 中所使用的 Kubernetes 服务镜像默认来自 Google 的 `gcr.io` 镜像仓库，而这个仓库在国内的访问存在问题。虽然还没有办法简单地将它替换掉，不过从代码的变化情况上看，这个最初被硬编码在“`images.go`”文件的仓库地址目前已经被转移到与配置相关的“`defaults.go`”文件里，将来很可能会变成一个可供用户配置的选项。

通过 `kubeadm` 构建的集群默认并没有部署 Kubernetes 的 Dashboard 操作界面，也没有配置容器间跨节点通信所需的网络。不过，这些工作并不复杂。在 Master 节点上执行以下命令，Kubernetes 将下载 Calico 网络工具，并自动完成集群节点互联所需的配置，如下所示。

```
$ kubectl apply -f \
http://docs.projectcalico.org/v2.4/getting-started/kubernetes/installation/host
ed/kubeadm/1.6/calico.yaml
```

关于 Calico 的原理和跨节点网络的实现将在第 6 章中详细描述。

观察 `kubectl get pods --all-namespaces` 命令的输出，当 `kube-dns` 的 Pod 状态从 `ContainerCreating` 变成 `Running`（这个 Pod 的正常运行依赖于跨节点网络）就表明容器网络已经能够使用了。接下来可以开始部署 Dashboard，执行以下命令。

```
$ wget https://rawgit.com/kubernetes/dashboard/master/src/deploy/
kubernetes-dashboard.yaml
$ echo ' type: NodePort' >>kubernetes-dashboard.yaml
$ kubectl create -f kubernetes-dashboard.yaml
```

第二条命令是让 Dashboard 服务使用 `NodePort` 的方式对外暴露访问入口，`NodePort` 是 Kubernetes 中的一种映射容器端口到特定节点端口的服务类型，关于 Kubernetes 的服务类型将在第 3.3 一节中详细介绍。接下来使用 `kubectl describe` 命令输出关于 `NodePort` 的端口号信息，如下所示。

```
$ kubectl describe svc kubernetes-dashboard --namespace=kube-system
... ..
NodePort:      <unset> 31333/TCP
... ..
```

例如，此时输出端口号为 31333，只需用浏览器访问集群中的任意 Node 节点 IP 地址加上 31333 端口即可打开 Dashboard 界面，如图 3-4 所示。

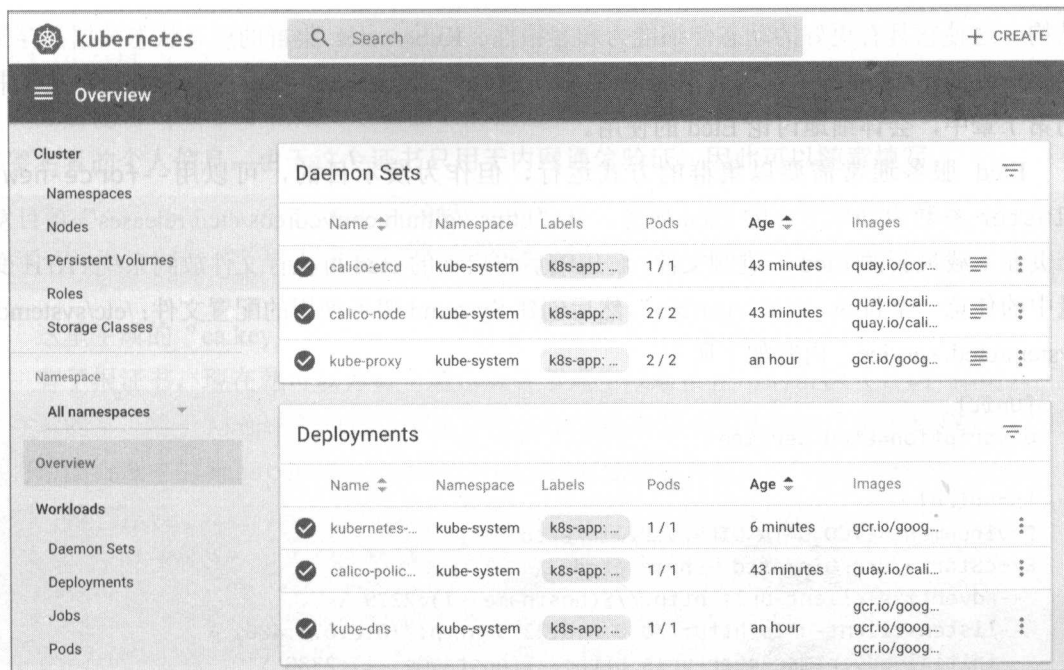


图 3-4 Kubernetes 的 Dashboard 界面

3.2.3 理解 Kubernetes 集群的部署过程

与 SwarmKit 将所有功能封装在一个 Docker 二进制文件的做法不同，Kubernetes 实际上是由许多独立的进程相互协作运行的。

除了官方主推的 Minikube 和 kubeadm 工具，社区中还有诸如 Kops、Kargo 等众多 Kubernetes 自动化部署项目。这些定制化的工具屏蔽了许多烦琐细节，使 Kubernetes 集群的部署变得十分便捷，然而这对于希望了解 Kubernetes 进程启动细节、定制运行参数，从而充分发挥 Kubernetes 能力的用户来说就显得过于笼统了。即使平时采用自动化方式部署集群，认识 Kubernetes 每个进程的真实启动过程在处理集群故障时也是很有帮助的。

下面以采用了 Systemd 管理系统进程的 Linux 为例（比如 Ubuntu 16.04、CentOS 7 以上系统），讲解不借助额外工具部署 Kubernetes 集群的常规方式。

1. 部署 Etcd

Etcd 是 CoreOS 公司研发的一种分布式存储服务，从本质上说，它是一种分布式的键值数据库。但相比普通的 NoSQL 数据库，Etcd 采用 Raft 共识算法来实现动态 Leader 节点

结构，这使它具有更好的动态伸缩能力和容错性。Kubernetes 集群的所有状态数据都存储在集群外的 Etcd 服务中，因此部署 Etcd 服务是部署 Kubernetes 集群的基础前提。在本书的第 7 章中，会详细地讨论 Etcd 的使用。

Etcd 服务通常需要以集群的方式运行，但作为演示目的，可以用 `--force-new-cluster` 参数启动单节点的 Etcd 服务。从“<https://github.com/coreos/etcd/releases>”项目发布页面下载最新的 Etcd 二进制文件，解压缩后将启动的 etcd 可执行文件放到系统 PATH 变量中的任意一个目录，例如“/usr/bin”。然后创建 Systemd 服务使用的配置文件：`/etc/systemd/system/etcd.service`，内容如下所示。

```
[Unit]
Description=Etcd Service

[Service]
Environment=ETCD_DATA_DIR=/var/lib/etcd
ExecStart=/usr/bin/etcd --name etcd0 \
  --advertise-client-urls http://$(hostname -i):2379 \
  --listen-client-urls http://0.0.0.0:2379,http://0.0.0.0:4001 \
  --initial-advertise-peer-urls http://$(hostname -i):2380 \
  --listen-peer-urls http://0.0.0.0:2380 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster etcd0=http://$(hostname -i):2380 \
  --initial-cluster-state new \
  --force-new-cluster
Restart=always
RestartSec=10s
LimitNOFILE=40000
TimeoutStartSec=0

[Install]
WantedBy=multi-user.target
```

使用 `systemctl start` 命令在系统后台启动配置好的 Etcd 服务，如下所示。

```
$ sudo systemctl start etcd
```

2. 创建 TLS 证书

为了数据安全起见，Kubernetes 建议采用 HTTPS 协议进行集群服务之间的通信。kubeadm 工具创建集群时会自动生成密钥文件，在手动部署的方式下，可以用 `openssl` 命令生成这些密钥。

首先创建一个目录用来保存生成的密钥文件，如下所示。

```
$ mkdir cert
$ cd cert
```

然后使用 `openssl` 生成一个自签名的根证书，如下所示。其中 `-subj` 参与用于指定证书签名者的个人信息，由于这个证书只用于内网通信验证，因此可以随意填写。

```
$ openssl genrsa -out ca.key 2048
$ openssl req -x509 -new -nodes -key ca.key -subj "/CN=linfan.cluster"\
-days 5000 -out ca.crt
```

这里生成的“ca.key”和“ca.crt”文件就是根证书的私钥和公钥文件。

有了根证书，现在就可以为每个通信服务节点生成通信使用的密钥（公钥+私钥）。需要注意的是，用于 Master 节点服务端密钥签名中的信息必须和实际的主机名保持一致，应该将下面命令中的 `<master-server-hostname>` 部分替换为实际 Master 节点的 `hostname`。

```
$ HOST=<master-server-hostname>
$ openssl genrsa -out server.key 2048
$ openssl req -new -key server.key -subj "/CN=${HOST}" -out server.csr
$ openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key\
-CAcreateserial -out server.crt -days 10000
```

这个证书是给 `kube-apiserver` 服务使用的，用于保护 Kubernetes 关键 API 的调用。同时，由于在 Master 节点上还运行有其他一些 Kubernetes 服务，它们在通信中属于调用 API 的客户端，因此还需要在 Master 节点上再生成一对客户端密钥。生成客户端密钥的命令与服务端密钥完全一致，差别仅仅是密钥的文件名（可以任意取名）。将下面命令中的 `<server-hostname>` 部分替换为实际 Master 节点的 `hostname`。

```
$ HOST=<server-hostname>
$ openssl genrsa -out client.key 2048
$ openssl req -new -key client.key -subj "/CN=${HOST}" -out client.csr
$ openssl x509 -req -in client.csr -CA ca.crt -CAkey ca.key\
-CAcreateserial -out client.crt -days 10000
```

接下来还需要给每个 Node 节点分别生成一对通信密钥。只需将上面这个命令中的 `<server-hostname>` 部分替换为每个 Node 节点相应的 `hostname`，分别生成密钥文件即可。原则上应该为每个节点分别生成一对独立的密钥，但由于 Kubernetes 的通信并不对 Node 节点密钥的签名信息有效性进行验证，在非正式的环境中，可用随意的主机名信息生成一个密钥分发给所有 Node 节点，甚至直接复用 Master 节点的客户端密钥。

最后将生成的证书分别拷贝到 Master 和 Node 节点上，如下所示。Master 节点需要根证书公钥和服务端密钥、客户端密钥。

```
$ sudo mkdir /opt/cert/  
$ sudo cp ca.crt server.crt server.key client.crt client.key /opt/cert/
```

Node 节点需要根证书公钥和客户端密钥（假设密钥名字都是“client.key”和“client.crt”），如下所示。

```
$ sudo mkdir /opt/cert/  
$ sudo cp ca.crt client.crt client.key /opt/cert/
```

3. 启动 Kubernetes 服务

准备好证书以后就可以启动 Kubernetes 的服务了。首先在“<https://github.com/kubernetes/kubernetes/releases>”页面下载最新的 Kubernetes 发行版文件“kubernetes.tar.gz”。

Kubernetes 的发行版文件包含了为各种不同平台构建的文件，分别单独进行打包，需要经过两次解压才能获得特定平台（例如 AMD64）的预编译文件，如下所示。

```
$ tar xzf kubernetes.tar.gz  
$ cd kubernetes/server/  
$ tar xzf kubernetes-server-linux-amd64.tar.gz  
$ cd kubernetes/server/bin  
$ ls  
hyperkube  
kube-apiserver  
kube-controller-manager  
kubectl  
kube-dns  
kubelet  
kubemark  
kube-proxy  
kube-scheduler  
... ..
```

将这些可执行文件拷贝到系统 PATH 环境变量的目录中，如下所示。

```
$ sudo mv kubernetes/server/bin/kube* /usr/local/bin/
```

(1) 配置服务参数

为了让所有服务能够顺利地连接到 Kubernetes 的 API Server 服务，需要告诉这些服务密钥文件的位置，可以使用 `kubectl config` 命令来完成相应的配置。这个操作需要在 Kubernetes 集群的左右节点上分别执行。将下面命令中的<master-server-hostname>替换为实际的 Master 节点 hostname。


```
$ MASTER_SERVER=<master-server-hostname>

$ kubectl config set-cluster stuq \
--server=https://${MASTER_SERVER}:6443 --certificate-authority=/opt/cert/ca.crt
$ kubectl config set-credentials client \
--client-certificate=/opt/cert/client.crt \
--client-key=/opt/cert/client.key
$ kubectl config set-context stuq-context \
--cluster=stuq --namespace=default --user=client
$ kubectl config use-context stuq-context
$ kubectl config set preferences.colors true
```

该命令会在系统中生成“\${HOME}/.kube/config”文件，Kubernetes 的服务启动时会使用这个文件作为默认的配置信息来源。

需要强调的是，上述的 **MASTER_SERVER** 变量值必须使用与生成服务端密钥签名时一致的 hostname，而不能使用 Master 节点的 IP 地址，否则服务在启动时会出现连接 API Server 错误的情况。

(2) 启动 Master 节点的服务

在 Master 节点上必须启动的服务有 **kube-apiserver**、**kube-controller-manager** 和 **kube-scheduler**。对于使用 Systemd 作为系统进程管理工具的 Linux 系统，只需在“/etc/systemd/system/”目录为这三个进程分别创建服务配置文件。注意要将示例中的 **<etcd-server- hostname>** 和 **<master-server-hostname>** 部分替换为实际的 Etcd 服务和 Kubernetes API Server 服务所在的节点 hostname。

首先是“**kube-apiserver.service**”文件，内容如下所示。

```
[Unit]
Description=Kubernetes API Service

[Service]
Environment=ETCD_SERVER=<etcd-server-hostname>
ExecStart=/usr/local/bin/kube-apiserver \
--logtostderr=true \
--v=2 \
--allow-privileged=false \
--etcd-servers=http://${ETCD_SERVER}:2379 \
--insecure-bind-address=0.0.0.0 \
--insecure-port=8080 \
--service-cluster-ip-range=10.100.0.0/16 \
--admission-control=ServiceAccount,LimitRanger,ResourceQuota \
--secure-port=6443 \
```

```
--client-ca-file=/opt/cert/ca.crt \  
--tls-private-key-file=/opt/cert/server.key \  
--tls-cert-file=/opt/cert/server.crt \  
--cors-allowed-origins='.*'  
Restart=always
```

```
[Install]  
WantedBy=multi-user.target
```

这里为了方便访问,同时开启了使用 HTTP 协议的 8080 端口和使用 HTTPS 协议的 6443 协议,而在之前的 kubeadm 方式部署时,默认只开启 HTTPS 协议。

其次是“kube-controller-manager.service”文件,内容如下所示。

```
[Unit]  
Description=Kubernetes Controller Manager Service  
  
[Service]  
Environment=MASTER_SERVER=<master-server-hostname>  
ExecStart=/usr/local/bin/kube-controller-manager \  
    --logtostderr=true \  
    --v=2 \  
    --master=https://${MASTER_SERVER}:6443 \  
    --service-account-private-key-file=/opt/cert/server.key \  
    --root-ca-file=/opt/cert/ca.crt  
Restart=always  
  
[Install]  
WantedBy=multi-user.target
```

还有“kube-scheduler.service”文件,内容如下所示。

```
[Unit]  
Description=Kubernetes Scheduler Service  
  
[Service]  
Environment=MASTER_SERVER=<master-server-hostname>  
ExecStart=/usr/local/bin/kube-scheduler \  
    --logtostderr=true \  
    --v=2 \  
    --master=https://${MASTER_SERVER}:6443  
Restart=always  
  
[Install]  
WantedBy=multi-user.target
```


最后依次使用 `systemctl start` 命令启动这些服务即可，如下所示。

```
$ sudo systemctl start kube-apiserver
$ sudo systemctl start kube-controller-manager
$ sudo systemctl start kube-scheduler
```

(3) 启动 Node 节点的服务

在 Node 节点上必须启动的服务有 `kube-proxy` 和 `kubelet`。在每个 Node 节点的“`/etc/systemd/system/`”目录中分别创建这两个服务的服务配置文件。注意要将示例中的`<master-server-hostname>`和`<node-server-hostname>`部分替换为实际的 Kubernetes API Server 服务所在的节点和当前 Node 节点的 `hostname`。

首先是“`kube-proxy.service`”文件，内容如下所示。

```
[Unit]
Description=Kubernetes Proxy Service

[Service]
Environment=MASTER_SERVER=<master-server-hostname>
Environment=NODE_SERVER=<node-server-hostname>
ExecStart=/usr/local/bin/kube-proxy \
    --logtostderr=true \
    --v=2 \
    --master=https://${MASTER_SERVER}:6443 \
    --hostname-override=${NODE_SERVER} \
    --proxy-mode=iptables
Restart=always

[Install]
WantedBy=multi-user.target
```

然后是“`kubelet.service`”文件，内容如下所示。

```
[Unit]
Description=Kubelet Service

[Service]
Environment=MASTER_SERVER=<master-server-hostname>
Environment=NODE_SERVER=<node-server-hostname>
Environment=DNS_SERVER=<dns-server-hostname>
ExecStart=/usr/local/bin/kubelet \
    --logtostderr=true \
    --v=2 \
```

```
--api-servers=https://${MASTER_SERVER}:6443 \  
--address=0.0.0.0 \  
--port=10250 \  
--hostname-override=${NODE_SERVER} \  
--allow-privileged=false \  
--cluster-dns=${DNS_SERVER} \  
--cluster-domain=cluster.local  
Restart=always
```

```
[Install]
```

```
WantedBy=multi-user.target
```

这个文件中还有一个占位符`<dns-server-hostname>`，它应该被替换为 Kubernetes DNS 服务所在节点的 `hostname`。如果集群没有使用 `kube-dns` 服务，则 `kubelet` 服务的 `--cluster-dns` 和 `--cluster-domain` 参数都可以去掉。稍后将介绍 Kubernetes DNS 服务。

需要注意的是，由于在服务端密钥签名时指定的所属者使用的是 Master 服务器的主机名而非 IP 地址，其他 Kubernetes 服务在连接 `kube-apiserver` 服务时，参数中也必须使用 Master 服务器主机名（否则签名验证会出错）。但如果在集群的 DNS 服务器上没有专门的配置，在 Node 节点上直接使用 Master 节点的主机名肯定会出现“Unknown Host”错误。因此在执行启动服务前，可能需要先修改 Node 节点的“/etc/hosts”文件，在其中添加 Master 节点的主机名和 IP 地址的映射。

最后同样需要用 `systemctl start` 命令启动这两个服务，如下所示。

```
$ sudo systemctl start kube-proxy  
$ sudo systemctl start kubelet
```

（4）启动其他可选服务

在 Kubernetes 中的 `kube-dns` 组件是可选部署的，它提供了集群服务通过服务名称相互通信的功能，这对于某些微服务的场景以及其他需要在集群内部进行服务通信的场景都十分有用。在早期版本里，这个功能是由 Kubernetes 的一个额外 SkyDNS 插件来完成的，Kubernetes 从其 1.3 版本开始新增了这个 `kube-dns` 服务，用于替代原先的插件功能。

这个服务的方法同样可以通过 Systemd 部署和启动，在“/etc/systemd/system/”目录创建“`kube-dns.service`”文件，内容如下所示。

```
[Unit]  
Description=Kubernetes DNS Service  
  
[Service]
```

```
Environment=MASTER_SERVER=<master-server-hostname>
ExecStart=/usr/local/bin/kube-dns \
    --logtostderr=true \
    --v=2 \
    --dns-port=53 \
    --domain=cluster.local \
    --kube-master-url=https://<MASTER_NODE>
Restart=always

[Install]
WantedBy=multi-user.target
```

使用 `systemctl start` 命令启动它，如下所示。

```
$ sudo systemctl start kube-dns
```

4. 安装扩展插件和跨节点容器网络

跨节点网络和其他插件的部署可以参考 `kubeadm` 方式中的相应部分。

值得一提的是，在 Kubernetes 发行包的“cluster/addons/”目录中同样包含了许多内置的扩展插件，提供诸如图形操作界面、用户服务日志采集、节点性能数据监控等额外的功能。

以安装 Kubernetes Dashboard 插件为例，除了通过 `kubeadm` 方式介绍过的直接使用 Dashboard 项目 GitHub 仓库上的 yaml 文件安装，也可以用 Kubernetes 发行包“addons”目录中的 yaml 文件安装。方法也十分简单，仅仅需要注意 Dashboard 服务默认使用 localhost 作为 Kubernetes API Server 的连接地址，因此要对“addons”目录中的文件进行一点修改。

进入“cluster/addons/dashboard/”目录，使用任意编辑器打开“dashboard-controller.yaml”文件，找到 `image: gcr.io/google_containers/kubernetes-dashboard-amd64` 这行，在后面补上一行内容，如下所示。

```
args: ["--apiserver-host", "http://<master-server-addr>:8080"]
```

其中的<master-server-addr>应该改为实际的 Master 节点 IP 地址，再使用 `kubectl create` 命令启动这个目录下的所有 yaml 描述文件就可以了，如下所示。

```
$ kubectl create -f dashboard-controller.yaml
$ kubectl create -f dashboard-service.yaml
```

这种方式创建的 Dashboard 没有使用 NodePort 的方法暴露服务入口，然而它将自己注册到了 Kubernetes 集群的 API 路由中，只需打开浏览器访问“`http://<master-server-addr>:8080/ui`”就能打开 Dashboard 的页面。

另外，“部署 Kubernetes 集群”和“构建跨节点网络”两者之间没有必然的先后顺序。通常如果使用的是非 CNI 插件的网络工具，例如 OVS 或传统方式部署的 Flannel，建议在安装好 Docker 之后就实施，这样便于调试和尽早发现问题，而对于 Calico 这类提供了 CNI 插件安装方式的工具，大可在集群部署完成后再实施。不同节点之间的容器网络是否能够互联本身并不会影响在 Kubernetes 集群中部署服务，然而当用户在集群中部署服务时，如果服务调度到不同的节点上，而这些容器之间无法直接进行通信，就会影响在容器集群上部署的服务的可用性。

相比 kubeadm，使用 Kubelet 服务和容器来管理其他 Kubernetes 服务的方式，使用 Systemd 来管理 Kubernetes 的所有进程（包括 Kubelet）更符合传统服务部署的习惯，同时能够更好地展示服务部署过程的细节。两者在本质上是相似的。

3.2.4 验证集群可用性

通过以上任意一种方式部署完 Kubernetes 集群后，接下来可以在集群中部署一些服务来验证它的功能已在正常运转。

这个例子将使用 WeaveWorks 公司的微服务演示项目：WeaveSocks。这个在线商城的 Web 应用由 14 个独立部署的子服务（包括数据库服务）组成，每个服务负责一部分独立业务，例如购物车系统、订单系统、支付系统等。使用 Git 拷贝其代码仓库，然后通过 `kubectl create` 或 `kubectl apply` 命令执行部署即可，如下所示。

```
$ git clone https://github.com/microservices-demo/microservices-demo
$ kubectl apply -f microservices-demo/deploy/kubernetes/manifests
```

这个应用的前端服务同样使用 NodePort 对外暴露了访问入口，通过 `kubectl describe` 找到相应端口号，如下所示。

```
$ kubectl describe svc front-end
... ..
NodePort:      <unset> 31747/TCP
... ..
```

使用浏览器访问 Node 节点的这个端口，将看到一个出售各式袜子的电子商城网站，如图 3-5 所示。

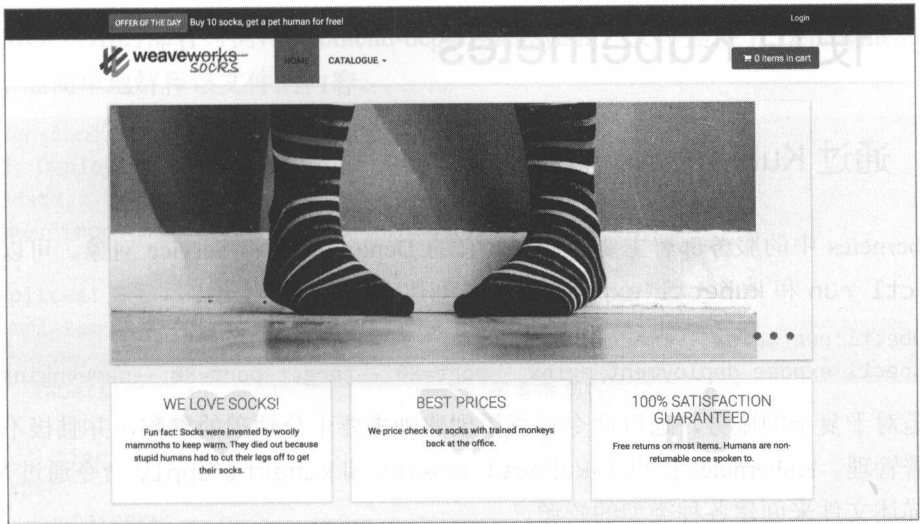


图 3-5 运行在 Kubernetes 中的 WeaveSocks 应用

此时，在 Dashboard 中切换到名称为“default”的 Namespace，就可以看到这个应用创建的各种 Kubernetes 资源对象，如图 3-6 所示。

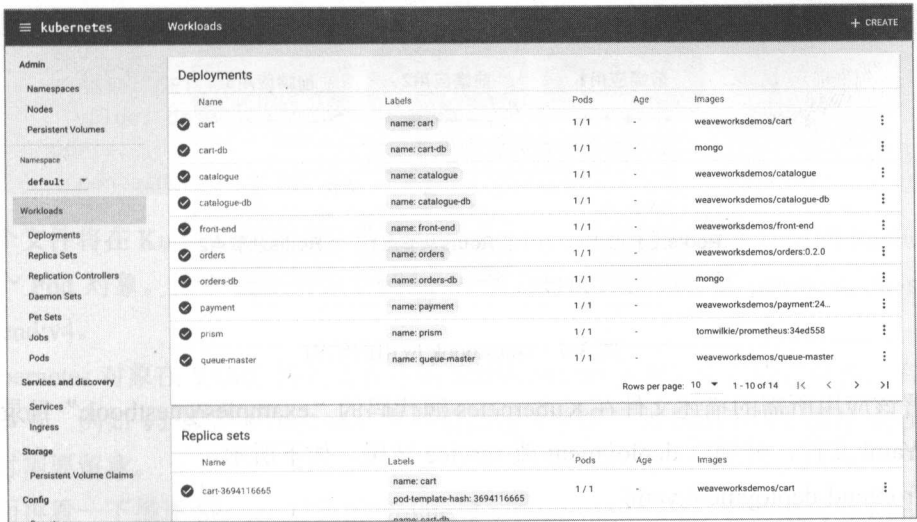


图 3-6 使用 WeaveSocks 应用创建的 Kubernetes 对象

最后，当不再需要这个应用的时候，可以使用 `kubectl delete` 命令将这个应用在集群中创建的所有服务全部销毁，如下所示。

```
$ kubectl delete -f microservices-demo/deploy/kubernetes/manifests
```

3.3 使用 Kubernetes

3.3.1 通过 Kubernetes 部署服务

Kubernetes 中的服务部署主要会使用到它的 Deployment 和 Service 对象。可以分别使用 `kubectl run` 和 `kubectl expose` 命令来创建它们，如下所示。

```
$ kubectl run nginx --image=nginx
$ kubectl expose deployment nginx --port=80 --target-port=80 --name=nginx-svc
```

但是对于复杂的服务，通过命令行直接创建则需要十分烦琐的参数，并且极不利于服务的部署管理。Kubernetes 提供了 `kubectl create` 和 `kubectl apply` 命令通过 Yaml 或 Json 的描述文件来创建各种类型的资源。

在 Kubernetes 示例中的 Guestbook 留言板应用展示了一个多服务构成的典型 Web 应用进行集群部署的过程，这个应用由一组 Web 页面服务和一个 Redis 集群组成，如图 3-9 所示。Kubernetes 提供了前端 Web 页面的负载均衡路由。

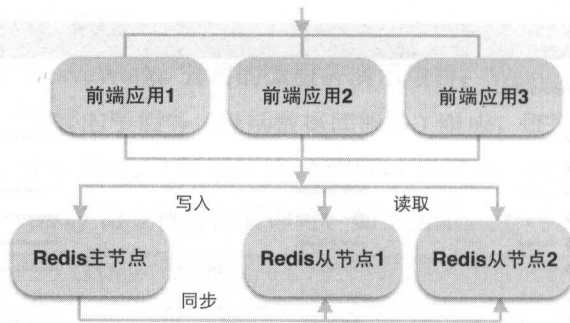


图 3-9 Guestbook 应用的结构

部署该应用所需的描述文件在 Kubernetes 源代码的“examples/guestbook”目录中，包含 6 个 Yaml 文件，分别以 deployment 或 service 结尾，如下所示。

- frontend-deployment.yaml
- frontend-service.yaml
- redis-master-deployment.yaml
- redis-master-service.yaml
- redis-slave-deployment.yaml
- redis-slave-service.yaml

以 Web 应用的部署为例,“frontend-deployment.yaml”文件提供了 Deployment 对象的描述,下面简单地解释该文件的内容。

```

apiVersion: apps/v1beta1      ←使用的对象描述规范版本
kind: Deployment              ←对象类型为 Deployment
metadata:
  name: frontend              ←生成 Deployment 对象的名称
spec:
  replicas: 3                 ←其中包含 3 个 Pod 副本
  template:                   ←这是每个 Pod 副本的模板
    metadata:
      labels:                 ←每个副本都具有两个标识标签
        app: guestbook
        tier: frontend
    spec:
      containers:             ←Pod 中的容器列表
      - name: php-redis
        image: gcr.io/google-samples/gb-frontend:v4
        resources:             ←该容器所需资源的描述
          requests:
            cpu: 100m
            memory: 100Mi
        env:                   ←声明该容器中的环境变量
        - name: GET_HOSTS_FROM
          value: dns
      ports:                   ←声明该容器会使用的端口
      - containerPort: 80

```

这个文件将在 Kubernetes 集群中生成一个 Deployment 对象、一个伴生的 ReplicaSet 对象和三个 Pod 对象。每个 Pod 对象中包含一个容器,使用的镜像是 gcr.io/google-samples/gb-frontend:v4。

Kubernetes 对象在 YAML 描述文件中的 apiVersion 参数值没有固定规律,有些是单纯的版本号,例如 v1 或是 v1beta1,有些前面包含前缀,如 batch/v1 或是 apps/v1,在使用时需要留意。

下面再看一下描述 Service 对象的“frontend-service.yaml”文件,它的结构与 Deployment 对象的 Yaml 文件如出一辙。

```

apiVersion: v1                ←使用的对象描述规范版本
kind: Service                  ←对象类型为 Service
metadata:
  name: frontend              ←生成 Service 对象的名称
  labels:                     ←该 Service 对象具有的标签

```

```
app: guestbook
tier: frontend
spec:
  ports:
    - port: 80
  selector:
    app: guestbook
    tier: frontend
```

←被选中 Pod 需要对外暴露的端口

←选择后端 Pod 的标签依据

Service 对象在 Kubernetes 中的作用是将集群中需要被外部访问的 Pod 暴露出来，本质上是一个使用 Iptables 实现的内置负载均衡。而它选择被路由 Pod 的依据是 Pod 的标签，在该描述文件中指定了两个标签属性，`app:guestbook` 和 `tier:frontend`。集群中所有同时具有这两个标签的 Pod 对象（即前面的 Deployment 对象创建的 Pod）都被作为这个 Service 的后端，图 3-10 展示了这几个对象之间的关系。

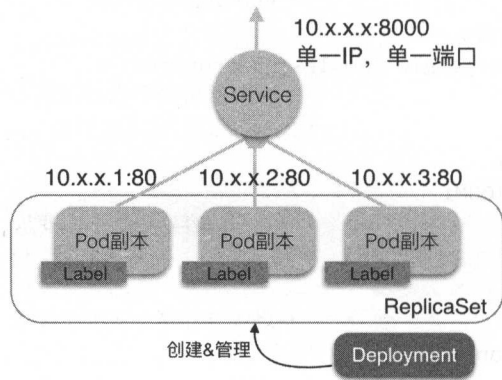


图 3-10 Kubernetes 对象之间的关系

需要指出的是，Kubernetes 的 Service 对象有四种类型，默认类型是 ClusterIP，这种情况下创建的负载均衡会使用内网的 IP 地址，只能从集群中节点访问。另外三种分别是 NodePort、LoadBalancer 和 ExternalName。其中 NodePort 指的是除了为 Service 生成内网 IP 地址，还会在每一个 Node 节点上占用一个端口，作为该服务的入口。而 LoadBalancer 则是在 NodePort 的基础上自动调用平台 API 为 Node 节点暴露的端口创建一个外部负载均衡器，但这个功能目前仅支持 AWS 和 GCE 等云平台，因此不具有通用性。ExternalName 类型服务的目的与前面几个相反，它将外部的服务地址暴露到集群内，并赋予一个能够通过集群 DNS 访问的服务名称。

为了能够在外部访问创建出的服务，需要在“frontend-service.yaml”文件的 spec 属性下面添加一个子属性 `type:NodePort`，如下所示。

```
spec:
  ports:
  - port: 80
    type: NodePort    ←声明服务类型为 NodePort
  selector:
    app: guestbook
    tier: frontend
```

其余的四个 Yaml 文件分别对应 Redis 的主节点和从节点的服务，其内容大同小异，这里不再赘述。

有了描述 Kubernetes 对象的文件后，使用 `kubectl create` 命令就可以将它们部署到 Kubernetes 的集群中了，如下所示。

```
kubectl create -f examples/guestbook
```

使用 `-f` 参数可以指定一个资源描述文件，也可以指定一个目录用于创建 Kubernetes 资源。当指定目录时，该目录下的所有 Yaml 文件（还有 Json 文件，但推荐使用 Yaml 文件）都会被读取。

创建完成后，可以使用 `kubectl get` 命令查看资源的运行状态，如下所示。

```
$ kubectl get pod
```

NAME	READY	STATUS	RESTARTS	AGE
frontend-88237173-06vc4	1/1	Running	0	8m
frontend-88237173-4nefq	1/1	Running	0	8m
frontend-88237173-hastq	1/1	Running	0	8m
redis-master-343230949-c3y0c	1/1	Running	0	8m
redis-slave-132015689-p2p46	1/1	Running	0	8m
redis-slave-132015689-x18ac	1/1	Running	0	8m

```
$ kubectl get rs
```

NAME	DESIRED	CURRENT	READY	AGE
frontend-88237173	3	3	3	9m
redis-master-343230949	1	1	1	9m
redis-slave-132015689	2	2	2	9m

```
$ kubectl get deploy
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
frontend	3	3	3	3	9m
redis-master	1	1	1	1	9m
redis-slave	2	2	2	2	9m

```
$ kubectl get svc
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
------	------------	-------------	---------	-----

frontend	100.74.140.190	<nodes>	80/TCP	9m
kubernetes	100.64.0.1	<none>	443/TCP	14m
redis-master	100.66.34.104	<none>	6379/TCP	9m
redis-slave	100.68.96.3	<none>	6379/TCP	9m

上述命令中的 `rs`、`deploy` 和 `svc` 分别是 `replicaset`、`deployment` 和 `service` 的简写，在 `kubectl` 命令的许多地方都可以使用这种简写规则，常用的简写表示如表 3-1 所示。完整的资源名称和简写列表可以在 `kubectl` 项目源代码的“`cmd.go`”文件^①注解里找到。

表 3-1 kubectl 命令行中的资源名称简写

资源名称	简 写
configmaps	cm
daemonsets	ds
deployments	deploy
events	ev
endpoints	ep
ingress	ing
limitranges	limits
nodes	no
namespaces	Ns
networkpolicies	netpol
pods	po
persistentvolumes	pv
persistentvolumeclaims	pvc
resourcequotas	quota
replicasets	rs
replicationcontrollers	rc
serviceaccounts	Sa
services	Svc

并不是所有的资源都有对应的简便写法，比如 `jobs`、`cronjobs` 和 `secrets` 都不能再简写了。可以试一试使用相似的命令，即用 `kubectl get` 加上这些资源名称，来获取它们的实例信息。

使用 `kubectl describe` 加上资源类型和资源名称，就可以获得特定资源实例的详细信息。通过这种方法能查找到 `frontend` 服务对外暴露的 `NodePort` 端口，如下所示。

① <https://github.com/kubernetes/kubernetes/blob/master/pkg/kubectl/cmd/cmd.go>

```
$ kubectl describe svc frontend | grep NodePort
... ..
NodePort:      <unset> 30286/TCP
... ..
```

打开浏览器，访问任意一个 Node 节点地址的 30286 端口，就会打开 Guestbook 的应用 Web 页面了，如图 3-11 所示。

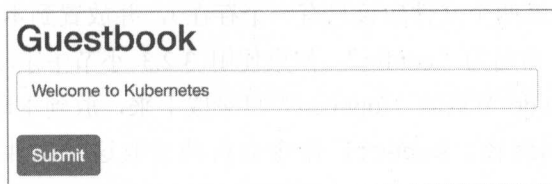


图 3-11 在 Kubernetes 集群中的 Guestbook 应用

使用 `kubectl delete` 命令可以删除指定的资源，如下所示。

```
$ kubectl delete deploy frontend redis-master redis-slave
deployment "frontend" deleted
deployment "redis-master" deleted
deployment "redis-slave" deleted

$ kubectl delete svc frontend redis-master redis-slave
service "frontend" deleted
service "redis-master" deleted
service "redis-slave" deleted
```

也可以用 `-f` 参数指定放有资源描述文件的目录，快速删除相应的资源，如下所示。

```
$ kubectl delete -f examples/guestbook
```

3.3.2 服务的在线更新和回滚

Kubernetes 支持对服务进行在线的扩容、缩容、升级和回滚，这意味着对应用在不停止对外提供服务的情况下完成版本和参数的修改。在 Kubernetes 文档的代码中提供了一个能直观地展示这个过程的可视化示例。

笔者将在本地计算机运行这个示例程序，同时运用 `kubectl` 工具提供的技巧，将 Kubernetes 的 API 代理到 `localhost` 地址上，以便这个程序能够从固定的地址读取集群的服务状态并展示出来。为此还需要为本地的环境做一些准备。

首先从 Kubernetes 发行版文件解压出来的 `platforms` 目录中找到用户本地操作系统相应

的 `kubectl` 命令行工具, 例如 Windows 系统可使用“`windows/amd64/kubectl.exe`”文件, MacOS 系统可使用“`darwin/amd64/kubectl`”文件, Linux 系统则是“`linux/amd64/kubectl`”。将其拷贝到系统的 `PATH` 变量指定的目录中, 这样接下来使用 `kubectl` 命令时就不用再输入完整路径了。

然后拷贝任意一个 Node 节点的“`~/.kube/config`”或“`/etc/kubernetes/kubelet.conf`”文件（取决于安装方式, 这两个文件应该只有一个存在）, 并放置到本地用户“Home”目录中的“`.kube`”子目录, 命名为“`config`”。如果使用 3.2.3 小节中手动创建的集群, 还应该将外置的证书文件从 Node 节点的“`/opt/cert`”目录取下来, 放到本地合适的位置, 并修改“`config`”文件中的证书路径。`kubectl` 命令会自动读取这个文件中的配置, 从而找到 Kubernetes 的 Master 节点。

由于 `kubeadm` 工具在生成 Kubernetes 证书的签名时会使用第一个网卡的 IP 地址作为签名的所有者, 但在有些公有云平台（例如 AWS）里, 服务器中的网卡只有内网的 IP 地址, 那么这个签名文件即使拷贝到本地也无法使用, 因此在使用默认的内网地址访问时, 会出现找不到目标主机的提示, 将配置改为公网地址则签名验证不过的尴尬情况。此时可以在集群内网的任意主机上直接执行下面的命令操作, 这样只会无法看到浏览器中展示执行过程的画面, 并不影响实际的执行结果。对于按照 3.2.3 小节手动步骤安装的集群, 还需要修改本地的“`/etc/hosts`”文件, 使得用户能够使用 Master 节点的主机名找到它的 IP 地址。

使用 `kubectl get node` 命令测试一下连通性, 如果一切正常, 此时应该已经可以通过本地的 `kubectl` 命令直接远程控制 Kubernetes 集群了。

最后, 从 Kubernetes 文档的源码仓库^①中将“`docs/user-guide/update-demo`”目录完整下载到本地（也可以直接下载整个仓库到本地）, 进入该目录, 执行 `kubectl proxy` 命令, 如下所示。

```
$ kubectl proxy --www=$(pwd)/local --port=8080
```

这个命令会监听本地的一个端口, 并将发送到此端口的请求代理到集群的 API 上。

其中的 `--www` 参数会将本地的指定目录（这里用的是放有网页文件的“`local`”目录）映射到代理服务的“`/static`”路径下面。此时使用浏览器打开“`http://127.0.0.1:8080/static/`”地址, 看到的是一个空白的页面。

在“`update-demo`”这个目录中有“`nautilus-rc.yaml`”和“`kitten-rc.yaml`”这两个文件。其中“`nautilus-rc.yaml`”是一个比较完整的 `ReplicationController` 资源描述文件, 包含了 Pod

^① <https://github.com/kubernetes/kubernetes.github.io>

副本的个数信息，如下所示。

```
apiVersion: v1
kind: ReplicationController    ←对象类型为 ReplicationController
metadata:
  name: update-demo-nautilus
spec:
  replicas: 2                  ←副本个数为两个，即运行两个相同的 Pod
  selector:
    name: update-demo
    version: nautilus         ←选择 Pod 的条件，应与以下的 Pod 标签一致
  template:
    metadata:
      labels:
        name: update-demo
        version: nautilus     ←每个 Pod 的标签
    spec:
      containers:
        - image: gcr.io/google_containers/update-demo:nautilus
          name: update-demo
          ports:
            - containerPort: 80
              protocol: TCP
```

“kitten-rc.yaml”文件的内容基本相同，只是修改了 ReplicationController 资源名称、镜像名称和 Pod 的名称，同时没有指定目标的 Pod 副本个数。

接下来先用“nautilus-rc.yaml”这个文件创建一个 ReplicationController，如下所示。

```
$ kubectl create -f nautilus-rc.yaml
```

正如它在资源描述文件中所描述的，Kubernetes 会自动为它创建两个 Pod，创建好以后，注意观察浏览器中的空白页开始出现如图 3-12 所示的两个小鱼图案。

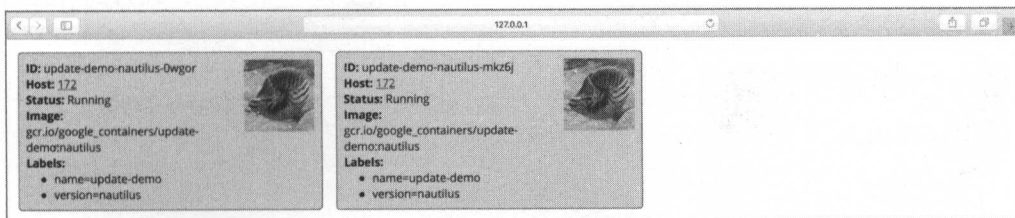


图 3-12 创建完成的两个 Pod

使用 `kubectl scale` 加上资源类型即可缩放 Pod 实例的个数，目前可以直接进行缩放的资源类型只有 `replicationcontroller`（可简写为 `rc`）和 `deployment`，如下所示。

```
$ kubectl scale rc update-demo-nautilus --replicas=4
```

此时浏览器中的小鱼数量也会变成四个，如图 3-13 所示。

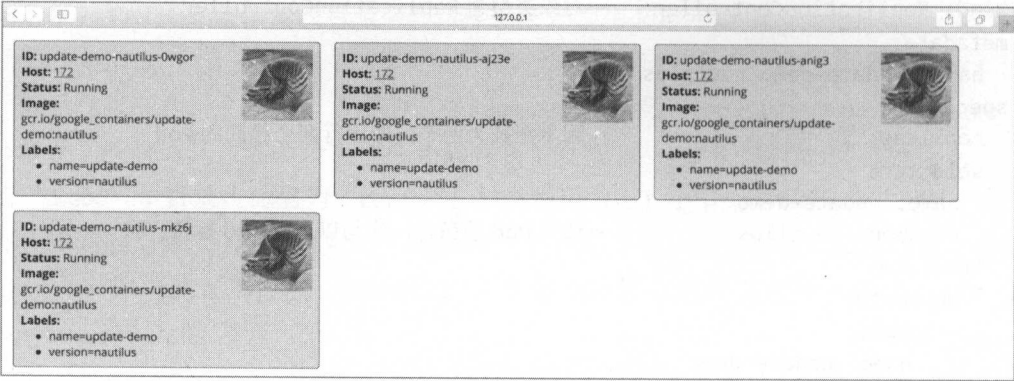


图 3-13 扩展为四个 Pod

Kubernetes 支持用滚动更新的方式实现服务的在线升级，即每次增加指定个数的新版本 Pod（默认为每次一个），然后删除相应个数的旧版本 Pod，持续这个过程，直到所有被管理的旧版本 Pod 都被更新完毕。相应的命令是 `kubectl rolling-update`，如下所示。

```
$ kubectl rolling-update update-demo-nautilus --update-period=3s -f kitten-rc.yaml
```

这个命令会以每三秒钟一个的速度替换当前运行的 Pod。图 3-14 展示了在升级过程中，浏览器上直观地表示 Pod 逐个替换的过程。在这个过程中，会有一个新旧版本同时存在的短暂时期，但由于始终保持着稳定的 Pod 总数目，应用对外提供的服务不会中断。



图 3-14 升级过程

最终所有旧版本的 Pod 全部被替换，得到如图 3-15 所示的结果。

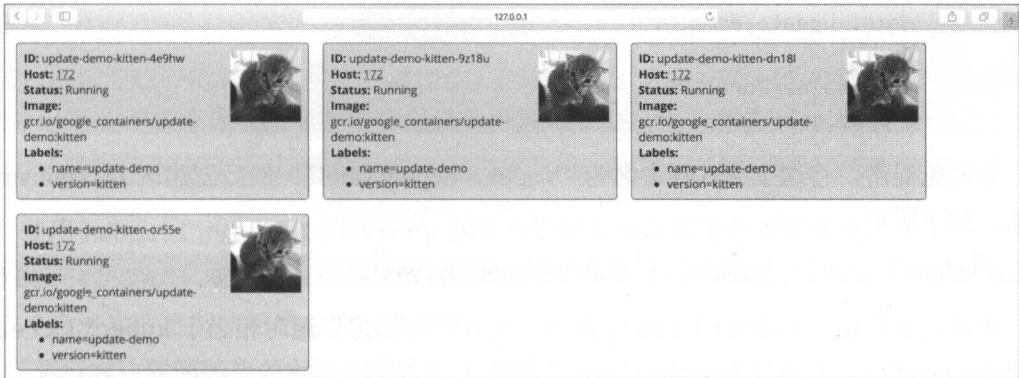


图 3-15 升级完成

如果在更新过程中发现配置有误，或由于其他原因使得升级过程无法完成，可以用“Ctrl+C”中断更新操作，并通过执行 `kubectl rolling-update --rollback` 进行回滚，命令如下所示。

```
$ kubectl rolling-update update-demo-nautilus update-demo-kitten --rollback
```

但如果升级完成后发现了问题，这个命令就无法进行回滚操作了，只能使用同样方法“升级”为旧版本。

接下来，尝试使用 Kubernetes 的 Deployment 资源替代 ReplicationController，修改 `nautilus-rc.yaml` 文件内容，如下所示。

```
apiVersion: apps/v1beta1          ←修改资源版本
kind: Deployment                  ←修改资源类型为 Deployment
metadata:
  name: update-demo-nautilus
spec:
  replicas: 2
  selector:
    matchLabels:                  ←增加一级 matchLabels
      name: update-demo
      version: nautilus
  template:
    metadata:
      labels:
        name: update-demo
        version: nautilus
    spec:
      containers:
        - image: gcr.io/google_containers/update-demo:nautilus
```

```
name: update-demo
ports:
- containerPort: 80
  protocol: TCP
```

为避免混淆，将这个文件重命名为“nautilus-deploy.yaml”，然后创建这个 Deployment 对象，如下所示。

```
$ kubectl apply --record -f nautilus-deploy.yaml
```

注意这里使用了 `kubectl apply` 命令，它在首次创建资源时相当于 `kubectl create --save-config`，会通过 `annotations` 注解将创建该资源所用的资源配置内容记录下来，以便查看。而 `--record` 参数会将这次执行的 `kubectl` 命令内容记录下来。下面这个命令打印出的 `kubectl.kubernetes.io/last-applied-configuration` 和 `kubernetes.io/change-cause` 注解内容即为该资源当前的配置信息和操作命令。

```
$ kubectl get -o yaml deployment/update-demo-nautilus
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: '{...}'
    kubernetes.io/change-cause: ...
... ..
```

Deployment 的资源升级不能使用 `kubectl rolling-update` 命令，应该直接将原先的资源文件内容修改，然后再次执行 `kubectl apply`。例如修改 `image` 属性的值为“`gcr.io/google_containers/update-demo:kitten`”，为了便于区分，可以将这个文件重命名为“`kitten-deploy.yaml`”，然后通过 `docker` 命令检查当前运行服务的镜像，如下所示。

```
$ kubectl apply --record -f kitten-deploy.yaml
$ docker ps | grep 'gcr.io/google_containers/update-demo'
```

此时将看到所有容器都已经被替换成标签为 `kitten` 的版本了。这个替换过程同样是逐个 Pod 进行的，类似于 `RollingUpdate`，然而使用者不能像 `kubectl rolling-update` 命令那样精确地控制每个 Pod 升级过程的间隔时间。

除了替换镜像，用户还可以替换其他的运行参数，例如修改 `replicas` 属性的值，然后执行 `kubectl apply` 命令，其效果相当于用 `kubectl scale` 命令对 Pod 副本个数进行扩缩。

如果觉得保留每个 Deployment 资源的原始文件以备将来修改的这种做法过于麻烦，可以直接用 `kubectl edit` 命令，它会自动读取当前资源的配置内容，并使用系统的命令行

编辑器（默认是“vim”，可以使用“KUBE_EDITOR”或“EDITOR”环境变量指定）打开。当用户对配置内容进行修改并保存退出后，Kubernetes 会自动应用新的配置。同样建议使用 `--record` 参数将该次操作的内容记录到资源对象中，如下所示。

```
$ kubectl edit --record deployment/update-demo-nautilus
```

使用 `kubectl rollout history` 命令可以查看指定资源的部署历史。由于在之前执行 `kubectl apply` 或 `kubectl edit` 命令时使用了 `--record` 参数，在显示的变更原因列中将包含这次变更所使用的命令，如下所示。

```
$ kubectl rollout history deployment/update-demo-nautilus
deployments "update-demo-nautilus"
REVISION CHANGE-CAUSE
1          kubectl apply --record -f nautilus-deploy.yaml
2          kubectl apply --record -f kitten-deploy.yaml
```

如果加上 `--revision` 参数指定某个版本号，还能看到该版本更详细的部署信息，如下所示。

```
$ kubectl rollout history deployment/update-demo-nautilus --revision=1
deployments "update-demo-nautilus" with revision #1
  Labels: name=update-demo
...
```

使用 `kubectl rollout undo` 命令就能回滚部署到上一个版本，如下所示。

```
$ kubectl rollout undo deployment/update-demo-nautilus
```

并不是所有的属性都可以在运行时进行更改，例如如果修改了 `Deployment` 描述文件的容器对外暴露的端口，或是加上一个 `type: NodePort` 的属性，那么试图使用 `kubectl apply` 命令更新的时候就会出错。此时就需要使用 `kubectl replace` 命令，如下所示。

```
$ kubectl replace --record -f nautilus-deploy.yaml
```

这个命令的效果相当于将相应的资源删除然后重建。不过 `kubectl replace` 的操作记录依然会出现在“rollout history”的列表中，并且可以用 `--record` 参数来添加操作命令的记录。

3.3.3 单次任务、定时任务和全局服务

`Job`、`CronJob` 和 `DaemonSet` 同样用于服务或事务的运行，并且同样基于 `Pod` 进行部署，

但它们与 Deployment 相比，分别具有一些独特之处。

Job 用于执行具有确定结束条件的单次任务，它们通常是一些运算类的事务，或由事件触发的批处理任务。这类任务不会长期在后台运行，并且在结束后会通过进程的返回值告知处理结果是否成功。

例如以下这个 Yaml 内容，是一个计算圆周率 Pi 小数点后 2000 位的任务，它会在计算完成后将结果输出到日志中，如下所示。

```
apiVersion: batch/v1          ←注意这个版本号
kind: Job                    ←资源类型是 Job
metadata:
  name: pi
spec:
  template:
    metadata:
      name: pi
    spec:
      restartPolicy: Never    ←即使出错也不需要重新执行
      containers:
      - name: pi
        image: perl
        command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
```

其中的 `restartPolicy` 是 Pod 的一个属性，对于 Job 中的 Pod，它的值可以是 `OnFailure` 或 `Never`，用于表示此任务在执行过程出错时是否需要重新执行。

将此文件内容保存为 “pi.yaml”，然后就可以用 `kubectl` 命令创建出相应的 Job 对象了，如下所示。

```
$ kubectl create -f pi.yaml
```

然后用 `kubectl get` 命令来获取 Job 对象的列表，如下所示。

```
$ kubectl get job
NAME      DESIRED  SUCCESSFUL  AGE
pi        1        0           8s
```

同时查看 Pod 列表会看到一个以 “pi-” 开头的对象，如下所示，在一段时间后这个 Pod 对象就不见了。

```
$ kubectl get pod
NAME      READY  STATUS   RESTARTS  AGE
pi-fuwfw  1/1    Running  0          20s
```

此时再次查看 Job 列表，将发现这个 Job 已经被标记为执行成功，如下所示。


```
$ kubectl get job
NAME          DESIRED  SUCCESSFUL  AGE
pi            1        1           8s
```

使用 `kubectl logs` 命令查看相应的 Pod，可以看到计算出来的 Pi 结果，如下所示。

```
$ kubectl logs pi-fuwfw
3.141592653589793238462643383279502884197.....
```

在更通用的场景下，Job 对象中的任务应该将运行产生的结果数据存放到数据库或其他公共存储空间里，以便其他服务能够方便地将其加以利用。

CronJob 是在 Kubernetes 1.4 中引入的新对象类型，它与 Job 对象相似，但并非只运行一次，而是周期性地运行。下面这个资源描述文件将每隔 10 分钟计算一次 Pi 的值，实际上 Pi 的值并不会随时间改变，这个例子仅仅是为了展示它与 Job 对象的差异。

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: pi
spec:
  schedule: "*/10 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: pi
              image: perl
              command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
              restartPolicy: OnFailure
```

将此文件保存为 “cron-pi.yaml”，然后用同样的方式创建它，如下所示。

```
$ kubectl create -f cron-pi.yaml
```

除了使用资源描述文件，也可直接用 `kubectl run` 命令创建 CronJob 对象，如下所示。

```
$ kubectl run pi --schedule="*/10 * * * *" --restart=OnFailure \
  --image=perl -- perl -Mbignum=bpi -wle "print bpi(2000)"
```

查看创建出的 CronJob 对象，如下所示。

```
$ kubectl get cronjob pi
NAME    SCHEDULE    SUSPEND  ACTIVE  LAST-SCHEDULE
pi      */10 * * * * False    0        ...
```

注意其中的 **LAST-SCHEDULE** 这列值，它会显示该任务最后一次执行的时间。

DaemonSet 对象部署的服务会在集群里的每个 **Node** 节点上都运行一个单独的 **Pod** 实例，它的应用场景主要是部署通用性的基础设施服务，例如监控工具或日志收集代理等。

下面这个例子会在集群的每个节点都部署一个监听 9090 端口的 **Nginx** 服务，它没有什么实际用途，只不过是便于验证和讲解。

```
apiVersion: extensions/v1beta1      ←注意版本号
kind: DaemonSet                    ←资源类型是 DaemonSet
metadata:
  name: nginx
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
              hostPort: 9090          ←这里指定了一个主机的端口
```

将以上内容保存为 “nginx-daemon.yaml” 文件，然后同样地使用 **kubectl create** 命令来创建，如下所示。

```
$ kubectl create -f nginx-daemon.yaml
```

同样可以用 **kubectl get** 命令来获取 **DemonSet** 对象的列表，如下所示。

```
$ kubectl get daemonset
NAME          DESIRED  CURRENT  NODE-SELECTOR  AGE
nginx         3        3        <none>         8s
```

这是在有 3 个 **Node** 节点的集群中显示的结果，该 **DemonSet** 对象会分别在每个节点上创建一个 **Pod**，可以用带 **-o wide** 参数的 **kubectl get** 命令来检查所有 **Pod** 对象运行的节点，如下所示。

```
$ kubectl get pod -o wide
NAME          READY  STATUS   RESTARTS  AGE  IP           NODE
nginx-3dh49   1/1    Running  0          49s  172.17.60.2  node-1
nginx-d83hx   1/1    Running  0          49s  172.17.46.3  node-2
nginx-eu36f   1/1    Running  0          49s  172.17.37.3  node-3
```

然后登录任意一个 Node 节点，执行 `curl localhost:9090` 后都会打印出 Nginx 默认主页的 HTML 内容。此时如果在集群中新增一个 Kubernetes 的 Node 节点，该节点上也会立即部署一个相同的 Pod，同时 Nginx DaemonSet 的 Pod 实例个数也变为 4，即证明 DaemonSet 对象会随着 Node 数量的变化自动调整目标实例的数量，以确保在所有节点上有且只有一个所属的 Pod 在运行。

3.3.4 持久化存储

Docker 提供了 Volume 特性来存储持久化的数据，主要是由于容器镜像对分层的支持，使用了 Copy-on-Write 特性的文件系统，此类文件系统的读写效率比普通的 Ext4 等文件系统更低，而 Volume 能够将一个非 Copy-on-Write 的普通目录挂载到容器里，从而获得更好的数据性能。此外，被挂载的目录还有可能来自分布式存储系统，从而将需要持久化保存的内容与容器实例分离，实现数据的灵活迁移。

Kubernetes 原生支持包括本地磁盘、NFS、Ceph 等诸多存储方式，在 Kubernetes 的文档中有一个完整的支持列表^①。其中最简单的两种类型是 `emptyDir` 和 `hostPath`，它们都表示使用本地的磁盘存储。`emptyDir` 类型的 Volume 会由 Kubernetes 在本地磁盘自动创建一个空目录并挂载，而 `hostPath` 类型的 Volume 则由用户指定挂载的本地目录。

值得一提的是，Volume 和容器一样，属于 Pod 管理的资源。这意味着在一个 Pod 中可以像定义多个容器那样定义多个 Volume，且这些 Volume 可以被同一个 Pod 里的所有容器共享，如下所示。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-log-collector
spec:
  containers:
    - name: nginx                ← 定义一个容器
      image: nginx
      volumeMounts:
        - mountPath: /var/log/nginx    ← 定义该容器挂载的 Volume
          name: log-volume             ← 挂载到容器中的位置
          name: log-volume             ← 引用 Volume 对象的名称
    - name: log-collector          ← 定义另一个容器
      image: custom-log-collector     ← 自定义的日志收集器镜像
```

① <http://kubernetes.io/docs/user-guide/volumes/#types-of-volumes>

```
volumeMounts:
- mountPath: /log/collector      ←挂载到不同的位置
  name: log-volume              ←挂载同一个 Volume
volumes:
- name: log-volume              ←定义一个 Volume
  hostPath:                     ←类型为 hostPath
    path: /tmp/log              ←hostPath 类型 Volume 的参数
```

如上面的例子所示，Volume 是被定义在与“containers”平级的 `volumes` 属性中的，每个 Volume 都有一个相应的名称，在容器中通过 `volumeMounts` 属性引用，不同的容器还可以分别挂载同一个 Volume 到各自指定的位置上。

在任意节点上使用 `kubectl get pod` 命令找到这个 Pod 的虚拟 IP 地址，使用 `curl` 访问它，屏幕上将打印 Nginx 服务默认首页的 HTML 内容，如下所示。

```
$ kubectl get pod nginx-with-log-collector -o yaml | grep podIP
podIP: 10.32.0.3
$ curl 10.32.0.3
<!DOCTYPE html>
... ..
```

此时查看本地的“/tmp/log”目录，会看到 Nginx 容器创建的两个日志文件通过 Volume 的挂载被放到了主机的目录中，而其中的内容正是刚刚访问的记录，如下所示。

```
$ ls /tmp/log
access.log error.log
$ cat /tmp/log/access.log
... "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"
```

除了这里演示的 `hostPath` 类型，其他的 Volume 类型在配置的属性上各有不同，但使用的方法是一样的，例如一个 Ceph RDB 的 Volume 配置可能如下所示。

```
volumes:
- name: rbdpd
  rbd:
    monitors: ["10.16.1.78:6789",
               "10.16.1.82:6789", "10.16.1.83:6789"]
    pool: kube
    image: foo
    user: admin
    keyring: /etc/ceph/keyring
    fsType: ext4
    readOnly: true
```

在 Pod 内直接定义 Volume 资源的方法十分简单，但在实际生产运用的时候可能会带来

一些问题，它使得服务的部署细节（Volume 在容器中的挂载点）与基础设施的实现细节（使用哪种类型的存储以及存储设施的参数）耦合在了一起。然而实际的情况是，即使是同一个服务，在不同环境部署时所用的存储方式和存储的参数（例如 Ceph 服务器地址）都不会是一样的，这将使得开发人员不得不为每个环境提供单独的 Yaml 文件用于部署，而这些文件大部分内容都是相同的。

Kubernetes 在设计之初就考虑到了这个问题，并引入了 PersistentVolume 和 PersistentVolumeClaim 两类特殊的资源。PersistentVolume 代表了一个环境中可用的 Volume 资源，而 PersistentVolumeClaim 是对资源使用需求的声明，它们在 Kubernetes 中可使用与其他资源相似的 Yaml 文件形式表示，如下所示。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv01
  labels:
    type: local
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/data/data01"
```

将上面所示的 PersistentVolume 资源的描述文件保存为 “demo-volumes.yaml”，然后使用 `kubectl create` 命令创建它，如下所示。

```
$ kubectl create -f demo-volumes.yaml
```

再创建一个使用该存储资源的使用声明，内容如下。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: demo
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

将它保存为 “demo-claim.yaml”，同样用 `kubectl create` 创建一个该资源的实例，如下所示。

```
$ kubectl create -f demo-claim.yaml
```

当一个资源使用声明被创建时，它会从当前环境中寻找能够适配需求的最小存储资源。由于只定义了一个大小为 10GB 的本地磁盘 Volume 资源，虽然该声明只需 3GB 的空间，依然会选择保存到先前创建出来的 Volume。PersistentVolumeClaim 对象会将被选中的 PersistentVolume 对象绑定，这样它就不能再被其他的 PersistentVolumeClaim 对象使用了。通过 `kubectl get` 命令可以查看它们之间的绑定关系，如下所示。

```
$ kubectl get pv
NAME LABELS CAPACITY ACCESSMODES STATUS CLAIM
pv01 type=local 10737418240 RWO Bound default/demo
$ kubectl get pvc
NAME LABELS STATUS VOLUME
demo map[] Bound pv01
```

在创建 Pod 时只需将挂载的 Volume 类型写成 persistentVolumeClaim，并指明使用的资源声明名称，Kubernetes 在创建该 Pod 的时候就会自动找到合适的 Volume 并进行挂载了，如下所示。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-log-collector
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - mountPath: /var/log/nginx
      name: log-volume
  volumes:
  - name: log-volume
    persistentVolumeClaim:
      claimName: demo
```

←类型为 persistentVolumeClaim
←指明引用资源的名称

3.3.5 配置存储

在一些设计方面比较好的 Cloud Native 架构中，通常会将服务应用程序本身与它的配

置信息分别进行管理，即让应用程序中不包含任何与特定运行环境配置相关的信息（例如数据库地址、密码等），而是在进程启动时从固定的地方获取，然后根据获取的配置去连接当前环境中的其他基础设施和依赖的服务。

这与传统的 Spring Cloud Config^①这类“集中式配置管理”项目不同，集中式的配置管理使得应用程序只需很少的配置参数：配置服务器的地址以及标识当前运行环境的名称标签。其他的配置信息都能够使用这两个参数从配置服务器上取得。然而这种方法并没有完全去掉配置参数。

一些分布式配置管理工具（比如 Etcd）通过在所有节点部署 Proxy 服务的方法解决了此问题：应用程序只需在启动时从 localhost 网络的固定端口读取配置，而无须关心当前是在什么环境运行。这使得开发者不用在代码仓库中放置任何体现部署环境的相关数据。但这种方法实际上会让服务的启动与 Etcd 等服务建立依赖，并且对于现有的很多应用程序，要全部改造成从外部服务获取配置信息依然不太现实。

为此，Kubernetes 设计了 ConfigMap 和 Secret 两种资源。这些资源可以由 Kubernetes 的系统管理人员创建，并在服务启动时转化为普通文件的形式加入目标服务的容器。从而既确保了服务开发人员无须知道运行环境的具体信息，又能兼容目前大多数 Linux 软件使用文件保存和读取配置的习惯。

ConfigMap 和 Secret 的对象的创建方法相似，两者的主要区别在于 Secret 对象中的数据是用二进制形式存储的，且所有文本需要先被序列化成 Base64 编码的字符串，这样即使到 Kubernetes 中将相应的存储对象信息打印出来，也无法很容易地看出 Secret 的内容，而 ConfigMap 对象则直接原样存储配置信息的明文内容，存储格式为普通字符串。和其他的 Kubernetes 资源一样，它们都可以用 Yaml 文件和“`kubectl create -f<文件名>`”的方式创建。下面就是一个包含数据库地址和用户名配置信息的“db.properties”配置文件作为 ConfigMap 对象的描述。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  color.good: purple      ←键值对数据
  color.bad: yellow
  db.properties: |-      ←配置文件数据
    db.addr: 10.134.1.1
    db.table: kube
```

^① <https://cloud.spring.io/spring-cloud-config>

由于 ConfigMap 中存储的内容通常是一些键值对,或是一个完整的 properties 配置文件, Kubernetes 提供了 `--from-literal` 和 `--from-file` 参数来简化配置信息的创建。首先创建一个内容如下的文件,并命名为“db.properties”。

```
db.addr: 10.134.1.1
db.table: kube
```

然后可以用 `kubectl create configmap` 命令来创建这个配置对象,如下所示。

```
$ kubectl create configmap db-config \
  --from-literal=color.good=purple \
  --from-literal=color.bad=yellow \
  --from-file=db.properties
```

同一个 ConfigMap 也可以包含许多个配置文件,只需使用多个 `--from-file` 参数。

Secret 对象的 Yaml 格式结构与 ConfigMap 很相似,但它有一个额外的 `type` 属性,该属性的值可以是 `Opaque`、`kubernetes.io/dockerconfigjson` 和 `kubernetes.io/service-account-token`。其中 `Opaque` 是最常见的一种,它表示声明的是一些应用的普通配置信息。`kubernetes.io/dockerconfigjson` 表示这是一个 Docker 本身的配置文件,通常用于拉取一些要登录才能使用的私有仓库 Docker 的镜像,在配置文件中添加用户信息。`kubernetes.io/service-account-token` 是为了让容器中的程序能够访问 HTTPS 保护的 `kube-apiserver` 服务,将通信证书 token 数据传送到容器内部的一种 Volume 对象。

此外,写在 Secret 对象 Yaml 文件中的所有数据值都需要使用 Base64 编码,由于多行文本进行 Base64 编码后也会成为单行数据,因此 Secret 对象的数据在创建时都可以用键值对形式表示。以下例子中的 `username` 和 `password` 值分别为 `admin` 和 `kube123`,但它们必须经过转码后才能被放置到 Secret 对象中被正常识别,如下所示。

```
apiVersion: v1
kind: Secret
metadata:
  name: db-user-pass
type: Opaque
data:
  username: YWRtaW4=
  password: a3ViZTEyMw==
```

Kubernetes 同样提供了 `kubectl create secret` 命令来简化 Secret 对象的创建。只需将每个需要存储的值以明文的方式存放到“以键名称命名”的文本文件中,如下所示。

```
$ echo -n "admin" > ./username
$ echo -n "kube123" > ./password
```

然后使用 `--from-file` 参数指定这些文件，如下所示。

```
$ kubectl create secret generic db-user-pass \
  --from-file=./username \
  --from-file=./password
```

如果要存储的键名称与文件名不一致，也可以用 “`--from-file=<键名称>=<文件名>`” 的格式，例如 `--from-file=username=./data.txt`，或者用 `--from-literal` 参数以明文的形式创建 Secret，如下所示。

```
$ kubectl create secret generic db-user-pass \
  --from-literal=username=admin \
  --from-literal=password=kube123
```

可以用 `kubectl get` 命令看到刚刚创建的对象内容，如下所示。

```
$ kubectl get configmap db-config -o yaml
apiVersion: v1
data:
  color.bad: yellow
  color.good: purple
  db.properties: |-
    db.addr=10.134.1.1
    db.table=kube
kind: ConfigMap
metadata:
... ..
```

```
$ kubectl get secret db-user-pass -o yaml
apiVersion: v1
data:
  password: a3ViZTEyMw==
  username: YWRtaW4=
kind: Secret
metadata:
... ..
```

创建好 ConfigMap 和 Secret 以后，就可以在 Pod 中将它们与部署的服务关联起来了。Kubernetes 支持使用环境变量和还原成文件这两种方式为服务提供配置信息。

下面这个 Pod 配置中使用 `configMapKeyRef` 和 `secretKeyRef` 分别读取了先前创建的 ConfigMap 和 Secret 内容，并赋值到环境变量中。

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: view-env-pod
spec:
  containers:
  - name: busybox-container
    image: busybox
    command: [ "/bin/sh", "-c", "env" ]
    env:
      - name: DB_COLOR_GOOD
        valueFrom:
          configMapKeyRef:
            name: db-config
            key: color.good
      - name: DB_COLOR_BAD
        valueFrom:
          configMapKeyRef:
            name: db-config
            key: color.bad
      - name: DB_SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: db-user-pass
            key: username
      - name: DB_SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-user-pass
            key: password
    restartPolicy: Never
```

←从 ConfigMap 读取

←从 Secret 读取

读取回来的 Secret 内容将以明文的方式存放，如下所示。

```
$ kubectl logs view-env-pod
... ..
DB_COLOR_BAD=yellow
DB_SECRET_USERNAME=admin
DB_COLOR_GOOD=purple
DB_SECRET_PASSWORD=kube123
```

另一种方式是将配置信息作为一种特殊的 Volume 挂载到容器，此时这些配置会以普通文件的形式存放到特定的目录。下面这个 Pod 配置分别挂载了 ConfigMap 和 Secret 对象，其中的 Secret 对象在容器中生成的配置文件内容同样会被还原成明文的形式，如下所示。

```
apiVersion: v1
kind: Pod
```



```

metadata:
  name: view-folder-pod
spec:
  containers:
    - name: busybox-container
      image: busybox
      command: [ "/bin/sh", "-c", "ls -R /etc/config*" ]
      volumeMounts:
        - name: config-volume-db
          mountPath: /etc/config
        - name: config-volume-db-secret
          mountPath: /etc/config-secret
  volumes:
    - name: config-volume-db
      configMap:                                ←挂载一个 ConfigMap 对象
        name: db-config
    - name: config-volume-db-secret
      secret:                                    ←挂载一个 Secret 对象
        secretName: db-user-pass
  restartPolicy: Never

```

这样的方式相当于能够直接生成应用程序的配置文件，对传统服务的部署比较友好。在 Kubernetes 发行版文件的“examples/mysql-wordpress-pd”目录中提供了一个使用 Secret 对象为 MySQL 数据库配置密码的例子。

值得一提的是，Kubernetes 的 Secret 对象除了作为普通应用的配置信息源以外，还可以用来存放 Docker Registry 的登录信息，用于辅助各个节点从私有仓库拉取 Docker 时进行自动登录，或者用于为运行在集群中且需要调用 Kubernetes 集群的 API 的用户服务提供 HTTPS 验证所需的 ServiceAccount 信息。其中拉取私有 Docker 仓库镜像的使用场景比较常见，下面简单介绍一下方法。

首先使用 Kubernetes 提供的 `kubectl create secret docker-registry` 命令来创建登录使用的 Secret 对象，如下所示。

```

$ kubectl create secret docker-registry dockerhub-key \
  --docker-server=hub.docker.com \
  --docker-username=linfan \
  --docker-password=dockerhub-password \
  --docker-email=linfan.china@gmail.com

```

然后在创建 Pod 时使用 `imagePullSecrets` 属性指定该 Secret 对象，如下所示。

```

apiVersion: v1
kind: Pod

```



```
metadata:
  name: private-image
spec:
  containers:
    - name: private-image
      image: "hub.docker.com/linfan/private-image"
  imagePullSecrets:
    - name: dockerhub-key
```

如果该 Pod 需要从多个私有仓库拉取镜像，可以分别创建 Secret 对象，然后在 imagePullSecrets 属性的值里将它们依次列出。

3.3.6 管理有状态的服务

容器在运行过程中总会或多或少地产生一些新的文件和数据。有些数据可以随时丢弃，例如服务使用的缓存文件和一些临时数据；有些数据虽然很重要，但采用了集中式的存储方式，例如存放到共享的文件服务器、外置的数据库或是对象存储系统；有些数据则需要服务的每个副本独享且不可以随意删除或调换顺序，例如数据库和配置服务（如 Etcd）的每个副本都应该有自己的独立数据目录。在 Kubernetes 中，有状态的服务指的正是最后这种需要稳定的持久化存储的服务，相应的服务对象类型最初被称为 PetSet（来源于第 1 章中提到过的宠物和牲畜的比喻），在 Kubernetes 1.5 版本中正式命名为 StatefulSet。

对于容器集群，有状态服务的挑战主要在于，通常集群中的任何节点都并非是 100% 可靠的，服务所需的资源量也会动态地更新改变。当节点由于故障崩溃或服务由于需要更多的资源而无法继续运行在原有节点上时，集群管理系统会为该服务重新分配一个新的运行位置，从而确保从整体上看，集群对外提供的服务功能不会中断。然而，对于有状态的服务，若采用本地存储，当服务漂移后数据并不会随着服务转移到新节点，重新启动的服务就会出现数据丢失的尴尬状况。若采用分布式的网络存储，通过 ReplicaSet 对象（即 Deployment 方式）部署的服务只能让所有副本共享相同的远程挂载目标，无法实现各个副本独立存储区域的目的。StatefulSet 的设计正好弥补了这种场景下的服务部署需求。

StatefulSet 同样通过 Pod 以及它的副本来组织容器，它会将每个属于自己的 Pod 副本编号，同时也将各个副本所挂载的存储卷编号，当发生容器漂移时，随着 Pod 副本被迁移到新的节点上，在 Pod 所属的容器启动时，又会自动找到属于自己的那个存储卷，重新挂载到新的节点上。这一切对于运行在容器内的服务程序而言，都是不可见却又自然而然发生的。下面以在 Kubernetes 部署高可用的 MySQL 服务集群为例，介绍 StatefulSet 的使用。

创建 StatefulSet 之前，需要先准备服务使用的网络存储数据卷。数据卷既可以由 StorageClass 自动添加，也可以使用 PersistentVolume 方式预先添加到集群里。为了更好地展示在 StatefulSet 中的一些隐含命名规则，这里采用静态 PersistentVolume 的方式，并假设使用 Ceph 提供网络存储能力。以下命令将创建名称为“demo_volume_0”的数据卷。

```
$ sudo rbd create demo_volume_0 --size 10240 \
  -m <Ceph Monitor 服务地址> --keyring /etc/ceph/ceph.client.admin.keyring
$ rbd feature disable demo_volume_0 exclusive-lock object-map fast-diff deep-flatten
```

第二条命令禁止了一些 Kubernetes 不需要的 Ceph RDB 特性，如果不这样做，可能会导致挂载数据卷失败。用同样的方法创建与所需 Pod 副本一样多的数据卷，依次命名为“demo_volume_1”“demo_volume_2”……关于 Ceph 和其他网络存储工具的知识，会在本书第6章中介绍。

然后创建同样多的 PersistentVolume 对象，这里不再对 Kubernetes 中 Ceph 类型的 PersistentVolume 对象配置做详细介绍，其结构大致如下所示。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: ceph-10gb-0
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  rbd:
    monitors:
      - <ceph-monitor-ip>
    pool: rbd
    image: demo_volume_0
    user: admin
    secretRef:
      name: ceph-secret
    fsType: ext4
    readOnly: false
  persistentVolumeReclaimPolicy: Recycle
```

然后创建同样多的 PersistentVolumeClaim。在 Kubernetes 文档中主要以 StorageClass 方式举例，由 Kubernetes 自动创建 PersistentVolumeClaim，对其中隐含的命名规律介绍一笔带过。但若是手动维护 PersistentVolumeClaim 对象，这一步中的名称定义非常重要，否则

会导致在创建 StatefulSet 的时候因无法找到 PersistentVolumeClaim 对象而失败。

正确的 PersistentVolumeClaim 对象命名规则是“StatefulSet 中的 volumeClaimTemplates 名称+StatefulSet 名称+序号”，例如下面这个 PersistentVolumeClaim 定义。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-mysql-0
  labels:
    app: mysql
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

隐含了之后定义的 StatefulSet 名称一定是“mysql”，且其使用的 volumeClaimTemplates 名称一定是“data”。继续创建足够多的 PersistentVolumeClaim，依次命名为“data-mysql-1”“data-mysql-2”……

接下来就可以创建 StatefulSet 和相应的 ConfigMap、Service 等对象了。Kubernetes 的官方案例提供了相应的资源描述文件，如下所示，这里不再详细说明。

```
$ kubectl create -f \
https://kubernetes.io/docs/tasks/run-application/mysql-configmap.yaml
$ kubectl create -f \
https://kubernetes.io/docs/tasks/run-application/mysql-services.yaml
$ kubectl create -f \
https://kubernetes.io/docs/tasks/run-application/mysql-statefulset.yaml
```

注意其中 StatefulSet 对象描述中的名称和数据卷声明模板的名称，如下所示。

```
apiVersion: apps/v1beta2
kind: StatefulSet
metadata:
  name: mysql
... ..
volumeClaimTemplates:
- metadata:
  name: data
... ..
```

与 Deployment 一次性创建出所有 Pod 不同，StatefulSet 创建 Pod 的过程是依次、有序

进行的，只有当前的 Pod 进入就绪状态才开始下一个 Pod 的创建。这样设计是因为许多有状态服务是区分主从身份的，Pod 依次启动使得用户有机会通过自动化手段使得服务的主节点在其他从节点启动前先就绪，如下所示。

```
$ kubectl get pods -l app=mysql
```

NAME	READY	STATUS	RESTARTS	AGE
mysql-0	2/2	Running	0	40s
mysql-1	1/2	ContainerCreating	0	19s
mysql-2	0/2	Pending	0	0s

此时如果其中的一个 Pod 或所在的节点意外崩溃，它将在其他节点上自动重建，并自动挂载上原先 Pod 的存储卷，实现有状态服务的故障漂移。

当对 StatefulSet 创建的服务集群进行扩缩容时，Kubernetes 总是确保按照 Pod 序列进行，例如将 MySQL 集群扩展为 5 个 Pod（若是自行维护存储资源，需预先准备足够的数据卷、PersistentVolume 和 PersistentVolumeClaim），如下所示。

```
$ kubectl scale statefulset mysql --replicas=5
$ kubectl get pods -l app=mysql
```

NAME	READY	STATUS	RESTARTS	AGE
mysql-0	2/2	Running	0	5m
mysql-1	2/2	Running	0	5m
mysql-2	2/2	Running	0	5m
mysql-3	2/2	Running	0	1m
mysql-4	2/2	Running	0	1m

然后缩小集群规模，如下所示。

```
$ kubectl scale statefulset mysql --replicas=3
$ kubectl get pods -l app=mysql
```

NAME	READY	STATUS	RESTARTS	AGE
mysql-0	2/2	Running	0	6m
mysql-1	2/2	Running	0	6m
mysql-2	2/2	Running	0	6m

可以看到，被删掉的 Pod 一定是末尾的“mysql-3”和“mysql-4”，首先创建并作为主节点的“mysql-0”总是安全的。当删除 StatefulSet 时，Kubernetes 会以相反的顺序依次销毁 Pod，当前一个 Pod 删除成功后，才继续下一个删除操作，最后被删除的将是“mysql-0”，如下所示。

```
$ kubectl delete statefulset mysql
$ kubectl get pods -l app=mysql
```

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

mysql-0	2/2	Terminating	0	8m
mysql-1	1/2	Terminating	0	8m
mysql-2	0/2	Terminating	0	8m

在 StatefulSet 被删除后，相应的存储卷和 PersistentVolumeClaim 并不会自动销毁。因此，即使 StatefulSet 已经被完全删除，只要再次创建它，其生成的每个 Pod 仍然能依次挂载原有的数据卷，并获得相应的状态数据。如果想释放这些资源，应该显式地执行删除操作，如下所示。

```
$ kubectl delete pvc -l app=mysql
$ kubectl delete pv ceph-10gb-0 ...
$ sudo rbd rm demo_volume_0 ...
```

3.3.7 健康检查

Kubernetes 的探针 (Probe) 功能为用户部署的服务提供了精确到容器的健康检查功能，以便在某些容器出现故障时能够及时对其进行路由屏蔽或重启。

探针有两种类型，分别是 LivenessProbe 和 ReadinessProbe，它们的使用方法基本一致，只是当探针检查到容器状态不正常时，LivenessProbe 探针会直接删除故障的容器，然后根据 Pod 中的 RestartPolicy 属性决定是否重新创建一个，而 ReadinessProbe 探针只会将该容器所属的 Pod 从所有使用它的 Service 对象的 Endpoint 里移除，即将通向该容器的路由断开。

探针判断容器是否健康的方式有三种：执行指定的用户脚本、访问特定的 HTTP 路径、检查特定的 TCP 端口。下面这个 Pod 展示了这些探针的用法。

```
apiVersion: v1
kind: Pod
metadata:
  name: probe-demo
spec:
  containers:
  - name: nginx-demo
    image: nginx
    livenessProbe:
      tcpSocket:
        port: 80
      initialDelaySeconds: 15
      timeoutSeconds: 1
    readinessProbe:
```

←探针是作用于指定容器的

←这是一个 LivenessProbe 探针

←判断方式是检查指定端口

←检查的时间间隔

←检查的访问超时时长

←这是一个 ReadinessProbe 探针


```

exec:                                ←判断方式是执行指定脚本
  command:
    - /usr/local/bin/health-checker
  initialDelaySeconds: 15
  timeoutSeconds: 1

```

每个容器最多可以有一个 LivenessProbe 和一个 ReadinessProbe。如果以上 Pod 中 nginx-demo 容器的 80 端口不能访问了，或者指定的 health-checker 脚本返回非 0 值，容器将被判断为不健康，并根据探针类型做出相应的处理。

由于 Nginx 实际上提供的是 HTTP 协议的服务，因此还可以将上述的 LivenessProbe 判断方式改为 HTTP 检查，即访问指定的路径，若返回非 2XX 的值，则认为容器不健康，配置如下所示。

```

livenessProbe:                       ←这是一个 LivenessProbe 探针
  httpGet:                           ←判断方式是检查 HTTP 服务
    path: /
    port: 80
  initialDelaySeconds: 15
  timeoutSeconds: 1

```

3.3.8 提供对外服务

在早期版本的 Kubernetes 中，如果希望将一个服务暴露到集群以外的地方访问，唯一的办法是使用 NodePort 或 LoadBalancer 类型的 Service 对象，但这两种方法都会在全部的 Node 节点额外占用一个网络端口，并且 LoadBalancer 只能在特定的云平台上使用。

Ingress 是在 Kubernetes 1.2 版本开始增加的特性，它允许用户创建一个固定的出口，通过负载均衡对外界提供服务。目前 Kubernetes 提供了基于 GEC 和 Nginx 的 7 层协议 IngressController 镜像，它们能够像普通的 7 层 HTTP 负载均衡服务器那样，在同一个端口上，通过访问的目的域名来转发到不同的后端服务地址。基于 Nginx 的 IngressController 不依赖于特定的云平台，下面介绍它的使用方法。

Nginx 的 IngressController 需要开启 Kubernetes 的 ServiceAccount 特性，因此首先需要确保启动 kube-apiserver 服务时的 `--admission_control` 参数值中包含 ServiceAccount。通过 kubeadm 方式部署的集群默认情况下此特性是开启的。

Kubernetes 提供了镜像 “gcr.io/google_containers/nginx-ingress-controller:0.8.3” 作为 IngressController 的实现，该镜像启动时需要通过参数指定一个“默认的服务后端”，也就是当请求的目标域名和路径与所有已注册的 Ingress 项目都不匹配时，就会使用这个服务处

理该请求。通常来说可以使用“gcr.io/google_containers/defaultbackend:1.0”镜像作为默认的后端服务使用，它封装了一个（除了/healthz 路径以外）永远返回 404 错误的 HTTP 服务进程。

可以使用如下所示的配置来创建一个默认的后端服务。

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: 404-backend          ← 默认后端 ReplicationController 名称
spec:
  replicas: 1
  selector:
    app: 404-backend
  template:
    metadata:
      labels:
        app: 404-backend
    spec:
      containers:
        - name: default-http-backend
          image: gcr.io/google_containers/defaultbackend:1.0
          ports:
            - containerPort: 8080      ← 稍后会在 Service 被映射到 80 端口
```

将其保存为“404-backend.yaml”文件，使用 `kubectl create` 创建这个 Replication Controller，然后使用 `kubectl expose` 命令为它生成一个 Service 对象，如下所示。

```
$ kubectl create -f 404-backend.yaml
$ kubectl expose rc 404-backend --port=80 --target-port=8080 --name=404-backend
```

然后就可以创建真正的 IngressController 服务了，将以下内容保存为“nginx-ingress-controller.yaml”文件。

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx-ingress-controller
spec:
  replicas: 1
  selector:
    k8s-app: nginx-ingress-lb
  template:
    metadata:
      labels:
```

```

k8s-app: nginx-ingress-lb
spec:
  containers:
  - image: |-
      gcr.io/google_containers/nginx-ingress-controller:0.8.3
    name: nginx-ingress-lb
    env:
      - name: POD_NAME
        valueFrom:
          fieldRef:
            fieldPath: metadata.name
      - name: POD_NAMESPACE
        valueFrom:
          fieldRef:
            fieldPath: metadata.namespace
    ports:
      - containerPort: 80
        hostPort: 80
      - containerPort: 443
        hostPort: 443
    args:
      - /nginx-ingress-controller
      - --default-backend-service=$(POD_NAMESPACE)/404-backend

```

←直接将容器端口映射到主机

使用 `kubectl create` 将这个 ReplicationController 创建出来，如下所示。

```
$ kubectl create -f nginx-ingress-controller.yaml
```

由于 IngressController 已经使用 `hostPort` 监听了所在节点的指定端口，因此不用再为它创建额外的 Service。

当 IngressController 的 Pod 启动后，其所在的节点就将成为 Ingress 配置的所有后台服务入口。通常不会希望这个 Pod 启动在一个随机的地方，因此需要固定它所运行的节点，具体的操作方法将在 3.3.10 这一小节中介绍。

接下来可以部署一些作为后端服务的容器了，这里采用 “`gcr.io/google_containers/echoserver:1.4`” 这个镜像，它会打印出所有请求的详细 HTTP 头和内容信息。通过参数让它监听 8080 端口，如下所示。

```
$ kubectl run echoheaders \
  --image=gcr.io/google_containers/echoserver:1.4 --replicas=1 --port=8080
```

然后将这个 Deployment 创建的 Pod 暴露为 Service，并且将容器的 8080 端口映射为 Service 的 80 端口，如下所示。

```
$ kubectl expose deployment echoheaders \
  --port=80 --target-port=8080 --name=echoheaders
```

最后，创建一个 Ingress 规则的描述文件，如下所示。

```
apiVersion: extensions/v1beta1
kind: Ingress                                ←类型是 Ingress
metadata:
  name: echomap
spec:
  rules:
    - host: foo.com                          ←指定域名
      http:
        paths:
          - path: /foo                        ←指定路径
            backend:
              serviceName: echoheaders        ←响应的 Service
              servicePort: 80                 ←Service 的端口
    - host: baz.com
      http:
        paths:
          - path: /bar
            backend:
              serviceName: echoheaders
              servicePort: 80
```

在此规则文件中定义了两个合法的请求路径，分别是“foo.com/foo”和“baz.com/bar”。可以将它们替换为实际属于你的域名，然后使用 `kubectl create` 命令创建这个 Ingress 对象，如下所示。

```
$ kubectl create -f ingress.yaml
```

以上述 Ingress 配置为例，假如用户确实具有 foo.com 和 baz.com 域名的所有权，可以将它们的解析地址修改为运行有 IngressController 的节点 IP。在演示中由于无法真实地修改这两个域名的解析地址，可以使用 `curl` 命令直接访问相应的节点 IP，并使用键为“Host”的 HTTP 头来告知 IngressController 要模拟访问的域名名称，如下所示。

```
$ curl <Nginx 节点 IP>/foo -H 'Host: foo.com'
```

IngressController 收到该请求后会根据第一个规则将它转发给 echoheaders 服务，并将 echoheaders 响应返回原始请求的所有 HTTP 头和内容显示到屏幕上。

同理，访问“baz.com”的“/bar”路径会得到类似的效果，而通过该域名的其他路径或其他解析到该主机的域名来访问，请求都将返回 404 错误。

3.3.9 多租户隔离和配额

Kubernetes 使用 Namespace 对象实现了租户隔离的功能。每位租户可以代表一位实际用户、一组特定服务、一种运行环境等含义，不同租户之间不能相互看到对方在 Kubernetes 中创建的资源。此外，Kubernetes 中对集群资源的限制也是基于租户设定的。

资源限制有两种类型：一种被称为 Resource Quota，它用于对整个 Namespace 内所有对象所用资源的总和进行限制；另一种被称为 LimitRange，它对指定 Namespace 中的每个 Pod 或容器所用的资源进行限制。

使用 Resource Quota 特性需要确保在启动 kube-apiserver 时的 `--admission-control` 参数值中包含 ResourceQuota。通过 kubeadm 创建的集群默认此功能是开启的。

Resource Quota 限制的资源包括计算资源，如 CPU、内存、虚拟资源、Pod、Service 等数量。完整的支持限制类型及其说明如表 3-2 所示。

表 3-2 可用的 ResourceQuota 类型及其说明

资源配额类型	说 明
cpu/requests.cpu	最大的 CPU 需求总量
memory/requests.memory	最大的内存需求总量
requests.storage	最大的磁盘存储需求总量
limits.cpu	最大的 CPU 限制总量
limits.memory	最大的内存限制总量
pods	可创建的非 Terminated 状态的 Pod 对象总量
replicationcontrollers	可创建的 ReplicationController 对象总量
services	可创建的 Service 对象总量
services.nodeports	可创建的类型为 NodePort 的 Service 对象总量
services.loadbalancers	可创建的类型为 LoadBalancer 的 Service 对象总量
configmaps	可创建的 ConfigMap 对象总量
secrets	可创建的 Secret 对象总量
persistentvolumeclaims	可创建的 PersistentVolumeClaim 对象总量
resourcequotas	可创建的 ResourceQuota 对象总量

以 Pod 数量为例，创建如下所示的资源描述文件，并命名为“pod-quota.yaml”。

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pod-quota
spec:
```



```
hard:
  pods: "2"
```

创建一个单独的 Namespace，并将 pod-quota 应用到该 Namespace 中，如下所示。

```
$ kubectl create namespace quota-example
$ kubectl create -f pod-quota.yaml --namespace=quota-example
```

使用 `kubectl describe` 能够查看指定的 Resource Quota 配置已经使用的量，如下所示。

```
$ kubectl describe resourcequota pod-quota --namespace=quota-example
Name:      pod-quota
Namespace: quota-example
Resource   Used   Hard
-----
pods       0       2
```

在该 Namespace 下创建一个 Pod 数量为 1 的 Deployment，如下所示。

```
$ kubectl run nginx --image=nginx --replicas=1 --namespace=quota-example
$ get pods --namespace=quota-example
NAME                                READY STATUS  RESTARTS AGE
nginx-3137573019-r2p2h             1/1    Running  0         12s
```

再次查看资源配额，将看到已经有一个配额被使用了，如下所示。

```
$ kubectl describe resourcequota pod-quota --namespace=quota-example
Name:      pod-quota
Namespace: quota-example
Resource   Used   Hard
-----
pods       1       2
```

然后尝试将它的个数扩展为四个，如下所示。

```
$ kubectl scale deployment nginx --replicas=4 --namespace=quota-example
$ kubectl get pods --namespace=quota-example
NAME                                READY STATUS  RESTARTS AGE
nginx-3137573019-8n8t3             1/1    Running  0         9s
nginx-3137573019-r2p2h             1/1    Running  0         1m
```

可以看到，由于资源配额的存在，实际只能创建两个 Pod 对象。

计算资源的配额比较特殊，一旦在 Namespace 中使用了此类资源配额，则创建 Pod 时必须指明所需的资源用量，否则会出现错误。创建如下所示的资源文件，将其命名为“cpu-mem-quota.yaml”。

```

apiVersion: v1
kind: ResourceQuota
metadata:
  name: cpu-mem-quota
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi

```

注意此次区别了“资源需求总量 (requests)”和“资源限制总量 (limits)”。前者指的是在调度资源时起作用，即只要目标主机上剩余计算资源满足 requests 提出的量，Pod 就可以被调度到该节点，可以将它视为相应 Pod 的最小资源需求。后者才是在 Pod 实际运行时能够使用的资源最大值，Pod 在运行时最多只会被分配到限制值内的资源量使用 `kubectl create` 命令创建这个资源配额对象。

```
$ kubectl create -f cpu-mem-quota.yaml --namespace=quota-example
```

下面的三个 Pod 创建操作只有第三个能成功。由于没有指定所需的资源用量，第一个命令会直接报错失败，在第二个命令中，Pod 需求的 CPU 量已经超过了所在 Namespace 能够使用的总 CPU 配额，只有第三个命令的 Pod 是符合限制的。

```

$ kubectl run nginx --image=nginx \
  --replicas=1 --namespace=quota-example

$ kubectl run nginx --image=nginx \
  --replicas=1 --namespace=quota-example \
  --requests=cpu=2,memory=256Mi \
  --limits=cpu=2,memory=512Mi

$ kubectl run nginx --image=nginx \
  --replicas=1 --namespace=quota-example \
  --requests=cpu=100m,memory=256Mi \
  --limits=cpu=200m,memory=512Mi

```

在使用 LimitRange 特性时，要确保在启动 kube-apiserver 时的 `--admission-control` 参数值中包含 LimitRange。通过 kubeadm 创建的集群默认此功能是开启的。

创建名为“limits.yaml”的资源描述文件，内容如下所示。

```

apiVersion: v1
kind: LimitRange
metadata:

```

```
name: demo-limits
spec:
  limits:
  - type: Pod
    max:
      cpu: "1"
      memory: 1Gi
    min:
      cpu: 200m
      memory: 6Mi
  - type: Container
    default:
      cpu: 300m
      memory: 200Mi
    defaultRequest:
      cpu: 200m
      memory: 100Mi
    max:
      cpu: "1"
      memory: 1Gi
    min:
      cpu: 100m
      memory: 3Mi
```

此文件分别对 Pod 和容器配额做了限制，其中容器配额还设置了 `default` 和 `defaultRequest` 这两项，它们将分别作为该 Namespace 中创建的容器的 `limits` 和 `requests` 资源的默认值。

新建一个 Namespace，然后为它应用这个 `LimitRange` 配额，如下所示。

```
$ kubectl create namespace limit-example
$ kubectl create -f limits.yaml --namespace=limit-example
$ kubectl describe limits demo-limits --namespace=limit-example
Name:      demo-limits
Namespace: limit-example
Type       Resource    Min      Max      DefReq   DefLimit  ReqRatio①
-----
Pod        cpu         200m     2        -        -         -
Pod        memory      6Mi      1Gi      -        -         -
Container  cpu         100m     2        200m     300m     -
Container  memory      3Mi      1Gi      100Mi    200Mi    -
```

① 为了便于阅读，此处对输出内容做了适当缩减。

尝试使用以下两个命令在该 Namespace 中创建 Pod（通过 Deployment 间接创建），只有第二个能够成功，因为第一个 Pod 所要求的 CPU 资源大于 LimitRange 配额限制的单个 Pod 最大 CPU 资源量。

```
$ kubectl run nginx --image=nginx \
  --replicas=1 --namespace=limit-example \
  --requests=cpu=2,memory=256Mi \
  --limits=cpu=2,memory=512Mi

$ kubectl run nginx --image=nginx \
  --replicas=1 --namespace=limit-example \
  --requests=cpu=100m,memory=256Mi \
  --limits=cpu=200m,memory=512Mi
```

3.3.10 集群的节点管理

在 Kubernetes 中，将节点也作为与其他资源相似的一种对象，同样可以使用 Yaml 文件来描述和管理。使用 `kubectl get node` 命令可以获得当前集群的节点信息，如下所示。

```
$ kubectl get node
```

NAME	STATUS	AGE
node1	Ready	1d
node2	Ready	1d
node3	Ready	1d

使用 `-o yaml` 参数，以 Yaml 文件格式保存其中一个节点对象的描述信息，如下所示。

```
$ kubectl get node node2 -o yaml > node2.yaml
```

使用以下两个命令中的任意一个，都能删除此节点。

```
$ kubectl delete node node2
$ kubectl delete -f node2.yaml
```

再次查看集群的节点信息，`node2` 节点已经不属于此集群了，如下所示。

```
$ kubectl get node
```

NAME	STATUS	AGE
node1	Ready	1d
node3	Ready	1d

此时可以用先前保存的 `node2` 节点描述文件重新添加这个节点，如下所示。


```
$ kubectl create -f node-2.yaml
$ kubectl get node
```

NAME	STATUS	AGE
node1	Ready	1d
node2	Ready	1d
node3	Ready	1d

在默认情况下，kubelet 进程启动时也会向其指向的集群 Master 节点注册，因此若是没有保存相应节点的描述文件，直接登录到目标节点上重启 kubelet 服务也能将该节点加回集群。

在 Kubernetes 调度 Pod 时，会根据节点的可用计算资源和 Pod 之间的关联关系，通过 Scheduler 服务（kube-scheduler 进程）计算出最佳的目标节点。但在某些情况下，会希望特定的 Pod 只运行在特定的节点上。此时可以使用节点的标签和 Pod 的 nodeSelector 属性来达到目的。

使用 kubectl label 命令为 node2 节点添加一个特定的标签，如下所示。

```
$ kubectl label nodes node2 disk-type=ssd
```

这样就为 node2 节点打上一个 disk-type=ssd 标签。每个节点可以具有多个不同的标签，每个标签也可以被添加到多个不同的节点上。在 Pod 的资源描述文件中使用 nodeSelector 属性可以将调度的范围缩小到所有具有该标签的节点，如下所示。

```
apiVersion: v1
kind: Pod
metadata:
  name: mongodb
spec:
  containers:
  - name: mongodb
    image: mongo
  nodeSelector:    ←节点选择属性
    disk-type=ssd
```

使用 -o wide 命令可以验证 Pod 的节点位置，如下所示。

```
$ kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	NODE
mongodb-3dfe8	1/1	Running	0	19s	node2

因为只有 node2 具有指定的标签，它获得了该 Pod 的运行权。需要注意的是，如果用户指定了 Pod 的 nodeSelector 条件，且集群中不存在包含相应标签的 Node 时，即使还有其他可供调度的 Node，这个 Pod 最终也会调度失败。

3.4 Kubernetes 包管理工具 Helm

3.4.1 Helm 简介

尽管使用 `kubectl` 和资源描述文件已经能够将相关的服务进行组合部署，但依然无法像 Docker Compose 那样方便地查看和管理由特定的一组服务部署出来的服务集合。也就是说，`kubectl` 提供了全局的通用服务操作能力，但并没有真正做到服务的编排管理。由于 Kubernetes 的流行，社区中很快就出现了一些相关的周边项目，其中较有影响力的是 Kompose^①和 Helm^②，这两款产品的代码仓库现在都被收纳到 Kubernetes 官方的 GitHub 组内。

Kompose 的思路是将 Docker Compose 的 Yaml 文件转换成一系列的 Kubernetes 资源描述文件，它实际上并未提供完整的编排管理功能。而 Helm 则是完全自立门户，创立了为 Kubernetes 量身定制的一套编排描述语法，同样使用 Yaml 格式表示。更进一步来说，Helm 所具有的能力不仅局限于对服务进行编排管理，它更像在 Kubernetes 集群上的服务集合包管理器，类似于主流 Linux 发行版里的 Apt 或 Yum 那类软件包管理器，为服务的部署提供发布版本管理、发布历史管理、指定版本回滚、服务发布仓库、参数化模板（多环境差异化部署）等丰富而实用的功能。

比较有意思的是，Helm 采用服务端 / 客户端架构，其中的服务端组件被称为“Tiller”，它是 Helm 客户端与 Kubernetes 交互的中介，并维护已经部署到集群中的服务状态。Tiller 服务端没有自己的数据库，而是使用 Kubernetes 的 ConfigMaps 存储状态信息。Helm 的客户端用于与用户交互，采用 gRPC 协议与 Tiller 通信，同时 Helm 会与保存服务包文件的仓库通信。Helm 将每个可以在目标集群上直接部署运行的服务集合包称为“Chart”。Helm 客户端从仓库里获得 Chart 部署文件和相关配置，然后通过 Tiller 将 Chart 部署到集群。

3.4.2 使用 Helm 管理服务

Helm 的客户端可以在 Linux、Mac 或 Windows 系统上运行。它只是一个单独的可执行文件，因此只需下载最新的发行包，解压后放到系统 PATH 变量所在的目录即可。例如 Linux

① <https://github.com/kubernetes/kompose>

② <https://github.com/kubernetes/helm>

的 64 位版本，如下所示。

```
$ wget \
https://storage.googleapis.com/kubernetes-helm/helm-v2.6.2-linux-amd64.tar.gz
$ tar xzf helm-v2.6.2-linux-amd64.tar.gz
$ sudo mv linux-amd64/helm /usr/local/bin/helm
```

第一次使用 Helm 前需要进行初始化，只需执行一条命令，如下所示。

```
$ helm init
```

如果集群启用了 RBAC 功能（譬如 kubeadm 默认配置部署的集群），则需要为 Helm 创建一个专用账号，如下所示。

```
$ cat <<EOF >helm.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: helm
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: helm
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: helm
  namespace: kube-system
EOF
$ kubectl create -f helm.yaml
$ helm init --service-account helm
```

这个操作会在目标 Kubernetes 集群上部署 Tiller 服务端，同时初始化 Helm 的配置。Helm 通过所在节点的“\$HOME/.kube/config”配置文件来获得目标 Kubernetes 集群的地址和连接方式，若初始化过程中出现“connection refused”等错误，则很可能是因为目标集群配置不正确。

用户无须自建或到网络上寻找 Chart 编排脚本，Helm 会像其他包管理工具一样，默认

从官方 Tiller 仓库^①查找并获取用户指定的 Chart 包。它和 Apt 比较像的是，会在本地缓存所有仓库的包资源索引，`helm repo update` 命令会更新这些索引，索引文件存放在“`~/helm/repository/cache`”目录，如下所示。

```
$ helm repo update
```

在安装任何 Chart 包之前，先来看 Helm 的 Chart 包结构，如下所示。

```
$ helm fetch stable/redis --version 0.10.2 --untar
```

这个命令会从“stable”仓库下载名称是“redis”、版本为 0.10.2 的服务 Chart 包，但并不安装它，而是解压到一个与服务同名的子目录，如下所示。

```
$ ls redis/  
Chart.yaml README.md templates values.yaml
```

使用 `helm lint` 可以检测指定目录中的文件是否符合 Chart 的约定结构，如下所示。

```
$ helm lint redis  
==> Linting redis  
Lint OK  
1 chart(s) linted, no failures
```

一个标准的 Chart 目录通常包含以下文件或子目录，如下所示。

- README.md: 项目描述文件，描述该 Chart 的用法和参数。
- Chart.yaml: 主配置文件，包含 Chart 的名称、版本和其他元数据信息。
- values.yaml: 模板文件中的使用变量的默认值。
- templates/: 在 Chart 使用的模板文件（即 Kubernetes 资源描述文件）。

有些 Chart 还会有使用协议和依赖关系等额外的文件和目录，如下所示。

- LICENSE: 使用协议说明文件。
- requirements.yaml: 用于存放当前 Chart 依赖的其他 Chart 的说明。
- charts/: 该目录中放置当前 Chart 依赖的其他 Chart。

使用 `helm install` 将指定目录中的服务集合部署到集群，如下所示。

```
$ helm install ./redis --name demo-redis
```

除了使用展开的包目录，Helm 也能使用未解压的 Chart 包或直接指定仓库中的 Chart 名称（以及版本）来部署服务，如下所示。

① <https://kubernetes-charts.storage.googleapis.com>

```
$ helm install redis-0.10.2.tgz
$ helm install stable/redis
$ helm install stable/mariadb --version 0.3.0
```

使用 `kubectl` 命令可以看到 Helm 创建的各种 Kubernetes 对象，如下所示。

```
$ kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
demo-redis-redis-589b449685-66jzx  1/1     Running   0           6m

$ kubectl get deployment
NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
demo-redis-redis  1         1         1            1           6m

$ kubectl get service
NAME           TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
demo-redis-redis  ClusterIP  10.107.32.172 <none>        6379/TCP  6m
```

在服务编排设计时往往会包含一些需要依据具体环境配置的内容，它们可以在 Chart 中以变量的方式体现，变量默认使用在 Chart 包中的“`values.yaml`”文件所提供的值来填充。用户可以在实际部署时用 `--set` 参数修改其中的一部分变量内容，如下所示。

```
$ helm install --set resources.requests.memory=512Mi mariadb
```

或是将要覆写的变量保存为单独的文件，部署时用 `-f` 参数加载它，如下所示。

```
$ helm install -f prod_values.yaml mysql
```

可以用 `helm ls` 列出所有已部署的包，部署到集群中的 Chart 在 Helm 中被称为“Release”，但这个词本身比较容易引起混淆，这里姑且将它叫作“Chart 实例”。如果部署时没有指定名称，Helm 会随机给 Chart 实例起一个名字。Chart 部署后，用 `helm status` 加上名字即可查看详细信息，如下所示。

```
$ helm ls
NAME           REVISION   UPDATED   STATUS   CHART           NAMESPACE
demo-redis     1          ...      DEPLOYED  redis-0.10.2    default

$ helm status demo-redis
LAST DEPLOYED: ...
NAMESPACE: default
STATUS: DEPLOYED
... ..
```

有专门的 `helm upgrade` 命令用于升级（实际上是用新的 Pod 替代旧的）指定的已部署的 Chart，如下所示。


```
$ helm upgrade demo-redis -f prod_values.yaml \
--set resources.requests.memory=1024Mi ./redis
```

Chart 实例是有部署版本的，从数值 1 开始，每次升级（即更新配置）都会递增。用 `helm history` 命令可查看指定 Chart 实例的更新历史，如下所示。

```
$ helm history demo-redis
```

REVISION	UPDATE	DSTATUS	CHART	DESCRIPTION
1	...	SUPERSEDED	redis-0.10.2	Install complete
2	...	SUPERSEDED	redis-0.10.2	Upgrade complete

还可以随时通过版本号将 Chart 回滚到任意一个历史版本，如下所示。

```
$ helm rollback demo-redis 1
```

若要删除已部署的 Chart，可使用 `helm delete` 命令，如下所示。

```
$ helm delete demo-redis
```

这个操作实际上只是逻辑删除，将服务停止并标记为已删除，并未真的清除相关数据。使用带 `-a` 或 `--all` 参数的 `helm ls` 命令可以看到已经被删除的 Chart 实例，如下所示。

```
$ helm ls -a
```

NAME	REVISION	UPDATED	STATUS	CHART	NAMESPACE
demo-redis	1	...	DELETED	redis-0.10.2	default

它的历史记录也还会存在，使用 `helm rollback` 命令可以恢复一个已经被删除的 Chart，如下所示。

```
$ helm history demo-redis
```

REVISION	UPDATED	STATUS	CHART	DESCRIPTION
1	...	SUPERSEDED	redis-0.10.2	Install complete
2	...	SUPERSEDED	redis-0.10.2	Upgrade complete
3	...	DELETED	redis-0.10.2	Deletion complete

```
$ helm rollback demo-redis 2
Rollback was a success! Happy Helming!
```

如果希望彻底删除一个 Chart 实例，可以在删除时加上 `--purge` 参数，如下所示，这样所有关于这个实例的数据就会被真正清除了。

```
$ helm delete --purge demo-redis
release "demo-redis" deleted
```


3.4.3 自定义 Chart

除了从网络上和各类仓库中获得现成的 Chart，用户也可以将自己的服务组合创建成新的 Chart 包，Helm 功能提供了一些方便的辅助功能来帮用户简化这类日常操作。

使用 `helm create` 命令可以快速创建出一个标准的 Chart 模板目录结构，它的参数是即将生成的 Chart 包名称，如下所示。

```
$ helm create demochart
Creating demochart

$ tree demochart/
demochart/
├── charts
├── Chart.yaml
├── templates
│   ├── deployment.yaml
│   ├── _helpers.tpl
│   ├── ingress.yaml
│   ├── NOTES.txt
│   └── service.yaml
└── values.yaml
```

除了之前介绍过的“Chart.yaml”“values.yaml”这些文件，在“templates”目录中还自动创建了一些文件，这些文件都不是必需的，其作用如下所示。

- NOTES.txt: 在安装 Chart 时自动显示的用户帮助文档通常会包含该 Chart 的使用和配置方法。
- deployment.yaml: 用来创建 Deployment 的资源描述示例。
- service.yaml: 用来创建 Service 的资源描述示例。
- ingress.yaml: 用来创建 Ingress 的资源描述示例。
- _helpers.tpl: 定义一些可以在 Chart 里引用的 Yaml 内容片段。

通常会直接删除“templates”目录里默认创建的这些文件，将真实的服务资源描述文件放进去，然后将需要复用或复杂的片段抽离到“_helpers.tpl”文件里，再创建一个“NOTES.txt”文件，写入自己 Chart 的基本使用说明。

在 Chart 的“templates”目录中的文件都是资源描述模板，之所以称为模板，是因为在其中可以嵌入许多“{{表达式}}”格式的变量数据和简单数据计算，例如下面这个模板。

```
apiVersion: {{ .Values.configmap.version }}
kind: ConfigMap
```

```

metadata:
  name: {{ .Release.Name }}-configmap
  labels:
    generator: helm
    date: {{ now | htmlDate }}
data:
  myvalue: "Hello World"
  {{- range $key, $val := .Values.favorite }}
  {{ $key }}: {{ $val | quote }}
  {{- end }}

```

其中使用了模板占位符、内置对象、函数、控制语句、模板变量等特性。

1. 模板占位符

占位符其实也是一类特殊的内置对象（Values 对象），表示引用一个在模板以外地方定义的数据，它是模板中最常见的语法单元。例如 `{{ .Values.configmap.version }}` 就引用了一个外部定义的、名为“configmap.version”的数据。占位符的数据来源主要有两种：一种是在 Chart 的“values.yaml”文件里定义的，对于在“charts”目录里的子 Chart，不但可以引用当前 Chart 的“values.yaml”文件的内容，还可以直接引用在父 Chart 的“values.yaml”文件中定义的内容；第二种是在执行 `helm install` 或 `helm update` 命令时，通过 `-f` 或 `--set` 参数设置的数据，这种用法已在前面的内容中介绍过。

2. 内置对象

在模板中有一些以“点”开头的占位符，例如表示当前 Chart 实例名字的 `{{ .Release.Name }}`，它们的值不需要用户自行定义，而是由 Helm 提供。除了包含 Chart 实例信息的 Release 对象，类似的内置对象还有包含 Chart 包信息的 Chart 对象、包含 Helm 和 Kubernetes 系统信息的 Capabilities 对象、包含当前模板信息的 Template 对象等。

3. 函数

Helm 提供了超过 60 种的常用函数，用于对数值内容进行加工。例如在 `{{ now | htmlDate }}` 中使用 `now` 函数可以得到当前的时间对象，而 `htmlDate` 函数会将这个对象转换为方便显示的日期文本。在模板中，函数前面的竖线表示一个管道操作，就是将前一步骤的结果作为后一步骤的输入。

这些函数中的绝大多数直接来自 Golang 语言的 Sprig 库，在这个库的文档^①里可以找到

① <https://godoc.org/github.com/Masterminds/sprig>

详细的列表，以下是其中一些比较常用的文本处理函数。

- `quote`: 给原始内容加上引号，并自动转义内容中原有的引号字符。
- `lower`: 将原始内容全部转换为小写字母。
- `upper`: 将原始内容全部转换为大写字母。
- `trim`: 去掉原始内容前后的空白字符。
- `nospace`: 去掉原始内容中的所有空格。
- `substr`: 截取原内容中的一段字符。
- `randNumeric`: 产生指定长度的随机数字串。
- `randAscii`: 产生指定长度的随机字符串。
- `indent`: 将原始内容缩进指定空格数。
- `replace`: 替换原始内容中的特定字符串。

4. 控制语句

Helm 的模板可以嵌入逻辑判断、循环等控制逻辑。例如根据内置对象的值改变模板的内容，如下所示。

```
{{ if eq .Values.server.type "redis" }}
specs: 2c8g
{{ else if eq .Values.server.type "mysql" }}
specs: 2c4g
{{ else }}
specs: 2c2g
{{ end }}
```

判断式中比较相等的“`eq`”、比较不相等的“`ne`”以及比较大小的“`lt`”“`gt`”等本质上也是函数。

对于比较复杂的数值对象，可以通过 `with` 语句简化对其中属性的访问。例如以下例子中的 `.name`、`.port` 实际上是 `.Value.server.name` 和 `.Value.server.port` 的简略写法。

```
{{- with .Values.server }}
name: {{ .name | upper | quote }}
port: {{ .port | default "5000" }}
{{- end }}
```

5. 模板变量

有些控制流需要和变量配合使用，例如下面这个循环语句。

```
{{- range $key, $val := .Values.env_variables }}
{{ $key }}: {{ $val | quote }}
{{- end }}
```

其中的`$key`、`$val`就是模板变量，而`:=`符号表示对变量的赋值。

6. 移除空白

由于使用了控制字段和占位符，有时可能会使生成的模板中出现不必要的空格或是空行，例如以下模板。

```
addr: {{ .Values.addr }}
{{ if ne .Values.hostname "" }}
hostname: {{ .Values.hostname }}
{{ end }}
port: {{ .Values.port }}
```

生成的描述文件会由于控制语句的存在而出现两个空行，如下所示。

```
addr: 10.10.10.10
                                     ←空行
hostname: demo
                                     ←空行
port: 5000
```

为此 Chart 提供一种移除空白的语法，符号是`{{-和-}}`。前者表示移除当前占位字段左侧的所有空白内容，包括换行符。后者表示移除当前占位字段右侧的所有空白内容，同样包括换行符。因此通常会在一些逻辑控制语句的地方看到`{{-}}`符号更常用，而不是`{{}}`。

同时移除左右两侧空白会使得上一行内容与下一行合并，如下所示。

```
addr: {{ .Values.addr }}
{{- if ne .Values.hostname "" -}}
hostname: {{ .Values.hostname }}
{{- end -}}
```

得到的结果可能如下所示。

```
addr: 10.10.10.10hostname: demo
```

7. 引用模板段

对于一些可能被复用的模板段，可以被抽离成“命名模板段”，使用 `define` 语句定义，在需要的地方使用 `template` 引用，如下所示。

```
{{- define "my_labels" }}
labels:
  generator: helm
  date: {{ now | htmlDate }}
{{- end }}
... ..
metadata:
  name: {{ .Release.Name }}-configmap
{{- template "my_labels" }}
```

这些命名模板段除了可以放在模板文件里，通常会被放到 Chart 的“templates”目录下的“_helpers.tpl”文件，从而可以在多个模板里共同引用。

3.4.4 Chart 仓库

除了使用官方的仓库，用户也可以创建自己的私有 Chart 仓库，并在 Helm 工具里添加和管理 Chart 仓库。

查看当前 Helm 客户端可知仓库列表的命令是 `helm repo list`，如下所示。

```
$ helm repo list
stable https://kubernetes-charts.storage.googleapis.com
local http://127.0.0.1:8879/charts
```

Chart 仓库实际上是一个普通的 HTTP 文件服务器，使用单层目录结构，存放许多 Chart 压缩包和一个“index.yaml”文件，后者是仓库的索引文件。几乎任何能提供 HTTP 文件访问的服务，例如 Apache/Nginx 服务、亚马逊 S3，甚至 GitHub Pages 服务都可以成为 Chart 仓库。默认安装后，Helm 包含两个 Chart 仓库，除了官方 stable 仓库外，还有一个 local 仓库。local 仓库可以让用户将自己的 Chart 包通过仓库统一管理起来，不过这个仓库并不是一直存在的，用户需要执行 `helm serve` 来创建一个默认监听“127.0.0.1:8879”地址的 HTTP 服务。如果希望仓库在整个集群都能被访问到，可以加上 `-address` 参数将监听地址改变为“0.0.0.0:8879”，如下所示。

```
$ mkdir charts
$ helm serve -address 0.0.0.0:8879 -repo-path ./charts
```

用 `helm package` 命令可将一个符合 Chart 规范的目录打包成 Chart 包文件，如下所示。

```
$ helm package demochart
Successfully packaged chart and saved it to: demochart-0.1.0.tgz
```


将自定义的 Chart 包放到仓库目录里，用 `helm repo index` 命令生成仓库的“index.yaml 索引”，如下所示。

```
$ mv demochart-0.1.0.tgz charts/  
$ helm repo index ./charts
```

然后重新生成 Helm 索引，就可以搜索并从仓库安装自定义的 Chart 包了，如下所示。

```
$ helm repo update  
$ helm search demochart  
NAME                VERSION DESCRIPTION  
local/demochart 0.1.0 A Helm chart for Kubernetes  
$ helm install local/demochart
```

可以用 `helm repo add <名称> <仓库 URL>` 来添加其他仓库到当前 Helm 客户端，例如添加官方的 incubator 仓库，如下所示。

```
$ helm repo add incubator \   
https://kubernetes-charts-incubator.storage.googleapis.com/
```

此外，Helm 还有一些重要但不算十分常用的高级特性，例如钩子任务、插件扩展等，逐一列举出来足以单独成章，为避免有喧宾夺主之嫌，这里不再进行详细展开。Helm 的文档^①里提供了更多细节和示例内容，有兴趣的读者不妨自行阅读。

3.5 本章小结

作为一个由庞大社区共同运作的项目，Kuberentes 在其设计方面处处体现着灵活和高度可扩展的特质，许多能力都以独立的资源对象提供。为了避免读者迷失在众多零散的细节里，本章先从典型的 Pod-Deployment-Service 资源组合入手，介绍服务部署、升级、管理的基本操作，然后依次引入其他重要的资源类型，逐渐串联成知识平面，并在最后一节介绍了基于 Kuberentes 的优秀服务编排管理工具 Helm。

相较于 SwarmKit 容器集群方案，Kuberentes 的学习曲线有明显的提升。但在理解和掌握其中的关键特性后，将对整个容器集群领域产生一览众山小的感觉，因此是十分值得深入钻研的进阶平台。

① <https://docs.helm.sh/>

第 4 章 Mesos 集群解决方案

Mesos 可以算得上开源界中的集群任务调度和控制系统鼻祖，与后来诞生的 Hadoop YARN 一度是分布式计算领域的两种首选任务调度方案。实际上，Mesos 很早就自己实现了容器方式的运行时隔离，所以在 Docker 容器出现后，Mesos 社区很快就提供了基于 Docker 的集群部署技术，之后又让自己的容器技术直接兼容 Docker 镜像，推出 Unified Container 特性。随着 Mesosphere 公司成立和基于 Mesos 的商业数据中心系统 DC/OS 发布，这个越发成熟的生态圈正在向人们展现其独有的魅力。本章将围绕 Mesos 和 DC/OS 解决方案，并将它们的核心功能逐一呈现给大家。

4.1 Mesos 和 DC/OS 概述

4.1.1 Mesos 项目的起源

Mesos 诞生于 2009 年，项目最初的创始人 Benjamin Hindman 当时正在加州大学伯克利分校的 AMP 实验室（全称是 Autonomous Mastery Prototyping Laboratory）攻读计算机科学博士学位。在那个时候，分布式计算还不是十分流行，出于研究目的，Benjamin 尝试将许多英特尔处理器芯片结合在一起，构建出了 64 核甚至是 128 核的机器。为了能够在这种多核心的 CPU 上均匀地分配执行任务，他自己编写了一个调度软件系统。这启发了他将这种任务调度形式扩展到多个独立计算机的想法。与此同时，Benjamin 的一些朋友，同在伯克利分校的 Andy Konwinski 和 Matei Zaharia，正在另一个实验室里开发一套跨数据中心运算的软件平台。在他们的帮助下，Benjamin 将自己的成果与这套平台进行结合，使之能够在单一的服务器集群上运行多个分布式系统，这便是 Mesos 的雏形。

为了演示基于 Mesos 之上创建的一个完整的新分布式系统能跑多快，他们又开发了一个分布式计算框架：Spark。这个后来赫赫有名的大数据计算平台，最初的版本只有 1300

行代码，耗时一个周末，它最初是运行在 Mesos 上的一个示例应用。

在 Mesos 计划进行一年后，Benjamin 和他的同事在 Twitter 进行了一次演讲，这次演讲成为了 Mesos 走向成熟的转折。有三位曾在 Google 工作过的 Twitter 成员在演讲后告诉 Benjamin，Mesos 的设计与 Google 所用的 Borg 系统十分相似，他们希望共同改进 Mesos，在 Twitter 重构一个更完美的数据中心调度系统。很快，Benjamin 成为 Twitter 的顾问，与这些前 Google 工程师们一起扩展了 Mesos 项目。而始终保持开源的 Mesos 项目也在 2011 年进入 Apache 基金会，成为 Apache 基金会旗下的顶级孵化项目。

后来，Andy 和 Matei 联合创立了 Spark 背后的商业实体公司 Databricks。而 Benjamin 本人则建立了 Mesos 背后的商业实体公司 Mesosphere，并随后推出了基于 Mesos 的商业数据中心操作系统解决方案：DC/OS。基于 Mesos 的数据中心已经在 Twitter、Netflix、AirBnb、Autodesk、Apple 等许多大型互联网公司得到了广泛使用，支撑着包括 Netflix 的海量在线视频业务和 Apple 公司的 Siri 语音助手那样的庞大后台服务。

4.1.2 Mesos 的结构

Mesos 是一套分布式集群管理器，旨在通过与不同任务之间共享资源的方式改进资源使用率。Mesos 则提供一种统一化资源视角，其涵盖全部集群节点并能以无缝化方式利用类似于单一计算机内操作系统内核的方式实现资源访问。因此，Mesos 被称为数据中心的内核机制，通过 Mesos 构建的数据中心应用，就像在有无限运算资源的巨型计算机上工作。

与 SwarmKit 或 Kubernetes 相比，Mesos 同样采用“主从结构”的部署方式。然而它提供了独有的“两级调度机制”，以便管理各种类型迥异的分布式应用程序。

第一级调度者是 Mesos Master 的守护进程，管理集群中所有运行 Mesos Agent 守护进程的节点。这些节点提供了运行任务的环境，最终执行任务的形式可以是 Java 应用、Hadoop 运算单元这样的传统程序，也可以是 Docker 容器这样的镜像化服务。

第二级调度是各种负责管理任务具体运行方式和状态的分布式系统（也被称为“服务框架”）。每个服务框架都包括调度器（Scheduler）和执行器（Executor）两个部分，其中调度器需要单独部署，并与 Mesos Master 保持通信，而执行器得到 Mesos 授予的计算资源后，在每个需要的计算节点上自动运行。调度器与 Mesos 层通过 API 通信，而不是直接跟物理机器打交道，这样的方式尝试解决的是资源的静态划分问题，最显而易见的好处是，用户可以在一批机器上运行多个不同的分布式系统，并更有效地动态划分和共享这些资源。另

一个重要原因在于它能够提供一个通用功能集（故障检测、分布式任务、任务启动、任务监控、结束任务、清理任务等），这样一来就无须每个分布式系统各自都重复地去实现这样一套逻辑。

Mesos Master 会与每个服务框架通信，并协调每个 Agent 节点，聚合计算各个节点的可用资源以及所有可用资源的总量，然后向注册到 Mesos 的服务框架发出资源邀约。服务框架根据运行任务的需求，选择接受或拒绝来自 Mesos 的资源邀约。一旦接受邀约，Mesos 会协调服务框架的调度器和 Agent 节点，在集群节点执行这个任务。多种不同的框架，比如 Marathon、Hadoop、Spark 的任务，可以在同一个节点上同时运行，如图 4-1 所示。

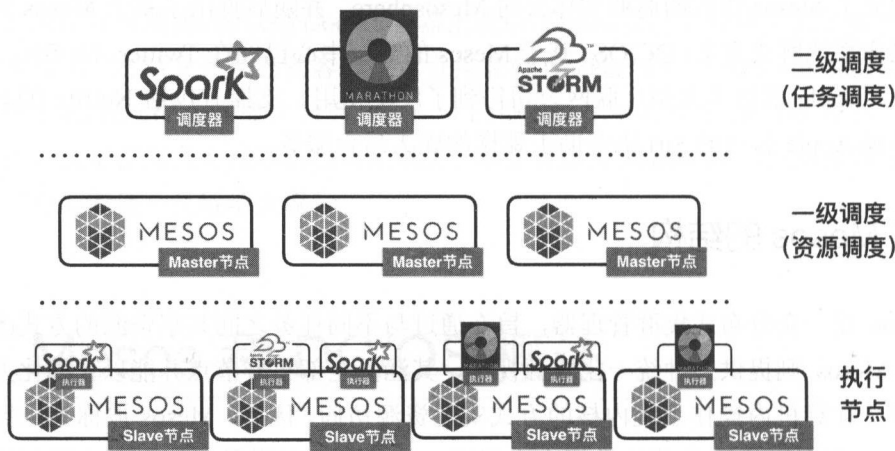


图 4-1 Mesos 的两级调度：资源调度和任务调度

当然，在一些通用的服务框架里也可以使用 Mesos 提供的资源进行更细粒度的调度，这便是所谓的“二次资源调度”。为了保证资源的相对隔离性，Mesos 利用 Linux 内核的隔离特性（包括 CGroups 和 Namespaces）提供了内置的容器支持，称为“Mesos 容器”，这是早在 Docker 还未出现之前 Mesos 就已经实现的自己的容器。现在 Mesos 同样支持使用 Docker 镜像来运行容器，并提供了两种运行方式。

一种是 Mesos 的 Docker 容器的执行组件（Docker Containerizer），它需要在各个 Agent 节点安装 Docker 后台服务，由 Mesos Agent 直接调用 Docker API 去创建和管理容器。另一种是在 Mesos 1.0 版本以后新增的“Unified Container”特性，它被 Mesos 容器赋予直接读取 Docker 和 OCP（OCI 标准）和 Rkt（AppC 标准）镜像的能力，这种方式不需要 Docker 后台服务的支持，而是像运行一个普通程序那样运行使用镜像打包的服务。本章的第 4.3.4 和第 4.3.5 小节将分别介绍这两种运行方式。

4.1.3 Mesos 的内部构成

Mesos 核心使用 C++编写，在内部通信中大量使用 Protobuf 协议，这是一种具有结构化语义的高效协议，对外则提供 HTTP 协议的接口。作为一个分布式系统，Mesos 原生支持高可用部署，只要集群中的大多数 Mesos Master 依然存活，整个服务就能得到保证。

Mesos Master 的主要职责如下所示。

- 对服务框架的注册、心跳监控和异常状态的处理，例如在框架的异常持续一段时间后，强行停止由该框架启动的任务进程，以避免出现失控的情况。
- 对 Agent 节点的添加、心跳监控、任务分配和异常状态的处理，例如将出现异常的 Agent 上正在执行的任务迁移到其他节点上继续运行。
- 对资源的管理，例如 Mesos Agent 服务定期上报每个节点上的可用资源数据，然后由 Mesos Master 服务把这些资源以 Resource Offer 的形式分配给集群中的服务框架使用。

Mesos Agent 主要负载管理任务的运行状态和服务框架的执行器 (Executor)。所有执行器都是在任务分配到 Agent 节点后，由 Mesos Agent 把执行器的二进制文件从 Master 节点提供的一个 URL 上获取下来，然后启动相应的执行器进程。Mesos Agent 节点可以动态地添加或减少，个别 Agent 节点故障一般不会产生对外部用户可见的影响，Mesos Master 会自动进行服务的重新调度。

服务框架 (Framework) 是 Mesos 进行任务调度的组件。在前面的两级调度模型中介绍过，Mesos 仅仅关心集群中资源的量以及把资源分配给谁，而由服务框架来决定这些资源怎么去使用。想象一下，在一个大型公司中有很多可用的硬件资源，现在由一组核心人员负责维护 Mesos 的集群，他们不断地在 Mesos 上添加资源和减少资源，而把服务框架执行的能力交给其他的业务组，各个业务组就可以使用自己的服务框架，放到整个 Mesos 集群上来执行。Mesos 执行各种各样的服务框架，而并不需要了解服务框架要做什么。

Mesos 核心与服务框架之间的关联是通过 Offer 和 Task 来体现的。

Offer 是 Mesos 资源的抽象，表示有多少 CPU、多少内存、多少可用磁盘空间等，在 1.0 版本以后还加入了 GPU 数量的支持，以增强对大数据计算任务的调度能力。Mesos 将从各个节点收集到的这些资源信息都放在 Offer 里，打包分配给服务框架，然后由服务框架来决定到底怎么用这个 Offer。

Task 实际是运行的小任务。有两大类 Task，一类是 Long Running Task，比如 Docker

的一个进程或者其他的进程。另外一类是 Batch 类型任务，这类应用非常广泛，比如 Map Reduce 这个小任务或者是定时任务。

4.1.4 DC/OS 数据中心操作系统

Mesosphere DC/OS 是 Mesos 的“核心”与其周边的服务及功能组件所组成的一个数据中心操作系统。作为一种大型数据中心的操作系统，DC/OS 使操作整个数据中心就像操作一台大计算机。这有点像 Linux 系统的工作模式，用户可以使用 Linux 内核自己构建操作系统，或者拿一个像 Ubuntu 这样的现成发行版。组成一个操作系统发行版的组件，如 RHEL 或 Ubuntu，通常会包含以下这些。

- Init 系统：一个守护进程（PID1），例如 Systemd、Upstart 或 Init.d，管理长时间运行服务，并负责当其管理的服务运行失败时根据配置自动重启。
- 定时任务系统：用于以特定的周期或在特定时间点触发任务执行，例如 Linux 中的 Cron 服务。
- 包管理：一个包格式（rpm、deb），相应的包管理器工具（yum、apt），以及一个基础仓库集合（base、main）和许多可选的第三方仓库。
- 命令行界面：一个 Shell，当用户登录时启动（bash、zsh）。
- 图形用户界面：一个用来观测和管理系统、可选的图形用户界面。

这些组件在 DC/OS 中都有相应的体现。它内置 Mesos-dns 这样的插件模块，提供相应的 SDK、CLI 和 GUI 交互界面，以及各种运行框架和软件包的仓库等设施，同时预装了 Marathon（分布式的 initd）和 Metronome（分布式的 cron）这样的通用任务调度框架。迄今为止，DC/OS 在其公有仓库上已经提供了包括 Hadoop、Spark、Jenkins、Kafka、Cassandra、HDFS 等在内的几十种服务组件包，用户也可以制作自己的组件包。

相比前几章介绍的 SwarmKit 或 Kubernetes，DC/OS 看起来要沉重和复杂许多。一方面，DC/OS 在部署时对系统硬件配置做了很多硬性的限制和要求。另一方面，DC/OS 中的许多组件分别使用了不同的编程语言来实现（这也是 DC/OS 生态圈开放性的体现）。不过，由于 DC/OS 很好地封装了底层的复杂性，对于使用者而言并不需要知道太多的细节。

对于想要真正理解 DC/OS 生态体系的读者而言，从其核心的 Mesos 入手是一种比较恰当的方式。本章接下来将从 Mesos 的部署开始，逐步介绍从零开始构建以 Mesos 为核心的集群的过程。

4.2 部署 Mesos 集群

4.2.1 部署 ZooKeeper

在 SwarmKit 和 Kubernetes 集群中, 为了确保集群的高可用性, 都使用了 Raft 协议来自动从多个 Master 节点中选举出一个公认的 Leader, 同时通过心跳监控和重新选举的方式来确保当 Leader 节点出故障时, 集群能快速与新的 Leader 达成共识, 从而避免群龙无首的混乱状态。两者稍有不同的是, Kubernetes 使用外置的 Etcd 服务来完成 Raft 协议选举的过程, 而 SwarmKit 索性把 Etcd 的 Raft 协议代码打包到了自己项目里。

然而在 2009 年 Mesos 刚诞生那会儿, Etcd 连影子都还没有, 那 Mesos 要怎么确保多个 Master 节点的高可用性呢? 前面说过, Etcd 的 Raft 协议实际脱胎于另一种更加古老的分布式一致性算法: Paxos。这个曾经因解决著名的拜占庭将军难题而闻名的算法可以追溯到莱斯利·兰伯特 (Leslie Lamport, 他也是著名的 LaTeX 排版系统的创作者) 在 1998 年发表的论文 *The Part-Time Parliament*。Paxos 算法十分晦涩复杂, 然而, 由于这篇论文的影响极大, 在当时还是出现了不少用 Paxos 算法的实现项目, 其中比较为人们所知的是开源的分布式键值存储系统 ZooKeeper 和 Doozer, 而 Mesos 正是使用 ZooKeeper 来记录 Master 节点地址, 并实现高可用的。

Paxos 算法的核心同样是全民投票, 只不过其中涉及的过程和角色要复杂得多。Mesos 利用 ZooKeeper 实现的分布式投票验证机制, 确保了即使在极端条件下, 只要还有半数以上成员存活, 集群依然可以选出唯一的 Leader。

实际上, 在没有 ZooKeeper 的情况下, 还是能够部署只有单个 Master 节点的 Mesos 集群的。不过作为通用方案, 建议即使始终使用 ZooKeeper 服务记录 Mesos Master 节点的地址。下面简单介绍 ZooKeeper 的安装步骤。

1. 准备 Java 运行环境

ZooKeeper 需要 JDK 1.7 版本以上的运行环境, 如果当前节点上没有安装, 可以在 <http://www.oracle.com/technetwork/java/javase/downloads/index.html> 页面下载和系统发行版一致的 OracleJDK 部署包。

或者也可以直接使用系统发行版的包管理工具安装 OpenJDK, 例如 Debian 和 Ubuntu 使用 apt 命令, 如下所示。

```
$ sudo apt install openjdk-8-jdk
```

RHEL 和 CentOS 发行版使用 `yum` 命令，如下所示。

```
$ sudo yum install java-1.8.0-openjdk
```

2. 下载 ZooKeeper

ZooKeeper 的下载页面是 [http:// ZooKeeper.apache.org/releases.html](http://ZooKeeper.apache.org/releases.html)，在国内建议使用清华大学的镜像站点，找到最新版本的服务安装包，下载后将其解压后拷贝到系统目录，如下所示。

```
$ wget https://mirrors.tuna.tsinghua.edu.cn/apache/zookeeper/zookeeper-3.4.9/zookeeper-3.4.9.tar.gz
$ tar xzf zookeeper-3.4.9.tar.gz
$ sudo mv zookeeper-3.4.9 /usr/local/zookeeper
```

3. 部署单节点的 ZooKeeper 服务

和 Etcd 一样，在测试使用的环境中，可以使用单个 ZooKeeper 提供存储服务。这样的 ZooKeeper 服务本身并不是高可用的，但它却能用于存储 Mesos Master 节点的信息，从而确保 Mesos 节点的高可用，如图 4-2 所示。

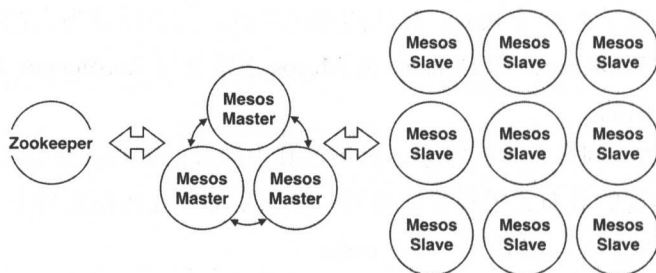


图 4-2 单节点 ZooKeeper 形成的高可用 Mesos 集群示意图

单节点 ZooKeeper 的部署比较简单，只需创建一个“`zoo.cfg`”配置文件，然后启动服务进程，下面举一个比较简单的配置示例。

```
$ cat << EOF | sudo tee /usr/local/zookeeper/conf/zoo.cfg
tickTime=2000
dataDir=/var/zookeeper
clientPort=2181
initLimit=5
syncLimit=2
EOF
$ sudo /usr/local/zookeeper/bin/zkServer.sh start
```

每个参数的含义如下所示。

- **tickTime**: ZooKeeper 服务器之间维持心跳的时间间隔, 以 ms 为单位。心跳包用于节点之间确认集群其他节点的存活状态, 也是当集群 Leader 失联后发起新选举的依据。
- **dataDir**: 顾名思义就是 ZooKeeper 保存数据的目录, 默认情况下, ZooKeeper 将写数据的日志文件也保存在这个目录里。在 ZooKeeper 服务第一次启动前, 此目录应该被预先清空以避免脏数据影响服务运行。
- **clientPort**: 用于与客户端通信的端口, 这个指定的 TCP 端口会被 ZooKeeper 监听, 并等待接受客户端的访问请求, 通常设置为 2181。
- **initLimit**: 在配置 ZooKeeper 服务器集群时, 若次节点为 Leader 节点, 表示等待的 Follower 节点初始化连接的最长时间。在这个例子中, 当超过 5 个心跳的时间 (也就是 tickTime) 后, 若 ZooKeeper Leader 节点还没有收到新加入 Follower 节点的心跳包, 则认为这个 Follower 节点连接失败。总的时间长度是 $5 \times 2000\text{ms} = 10\text{s}$ 。
- **syncLimit**: 这个配置项表示 Leader 与 Follower 之间发送消息, 请求和应答时间长度最长不能超过多少个 tickTime 的时间长度。在这个例子中, 总的时间长度就是 $2 \times 2000\text{ms} = 4\text{s}$ 。

需要注意的是, 使用单节点模式配置的 ZooKeeper 没有高可用副本, 所以如果此 ZooKeeper 服务节点出现故障, 整个服务就会终止。

4. 部署高可用的 ZooKeeper 集群

使用多个 ZooKeeper 节点构成高可用集群, 然后用这个 ZooKeeper 的集群实现 Mesos Master 节点的高可用, 可以形成完全高可用的 Mesos 集群, 如图 4-3 所示。

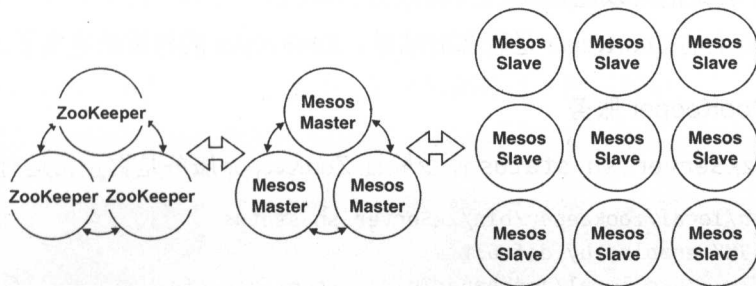


图 4-3 完全高可用的 Mesos 集群示意图

在这个结构中, ZooKeeper 集群的高可用是其自身使用 Paxos 协议投票选举直接保证

的，而 Mesos Master 集群则是借助 ZooKeeper 的 Paxos 投票功能为自己选出合适的 Leader。作为 Leader 的节点有权发起和确认该集群内所有数据的写入操作。

部署这样的 ZooKeeper 集群需要至少三个服务器节点，然后稍微修改每个节点上的“zoo.cfg”配置文件，使之能够相互识别，最后分别在每个节点的数据目录内创建一个保存当前节点序号的“myid”文件并启动服务进程即可。以下是一个使用三个节点的示例。

```
$ cat << EOF | sudo tee /usr/local/zookeeper/conf/zoo.cfg
tickTime=2000
dataDir=/var/zookeeper
clientPort=2181
initLimit=5
syncLimit=2
server.1=172.31.31.164:2888:3888
server.2=172.31.27.243:2888:3888
server.3=172.31.27.16:2888:3888
EOF
$ echo 1 | sudo tee /var/zookeeper/myid
$ sudo /usr/local/zookeeper/bin/zkServer.sh start
```

相比单节点的 zoo.cfg 内容，新增的部分是 server.X 配置项。其中的 X 是一个数字，数值需要与各个节点的“myid”文件内容相同，表示是第几号服务器。该参数的值分成三个部分，使用冒号分隔。第一部分是这个服务器节点的 IP 地址。第二部分表示的是这个节点与集群中的 Leader 节点交换信息的端口。第三部分表示的是万一集群中的 Leader 节点挂了，用来执行选举时节点相互通信的端口。它们通常是 2888 和 3888。

“myid”文件在各个节点上应该都不相同，其内容只有一个数字，并与 zoo.cfg 所列的节点编号一致。例如 IP 地址是 172.31.31.164 的节点内容为 1，IP 地址是 172.31.27.243 的节点内容为 2，其余的可以此类推。

当每个节点上的 ZooKeeper 进程都启动后，ZooKeeper 的部署就完成了。

5. 测试 ZooKeeper 服务

可以使用 zkServer.sh status 命令验证 ZooKeeper 服务是否正常运行，如下所示。

```
$ sudo /usr/local/zookeeper/bin/zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Mode: leader
```

或者使用 zkCli.sh 工具连接到 ZooKeeper 服务验证数据读写是否正常，如下所示。


```
$ /usr/local/zookeeper/bin/zkCli.sh -server 172.31.31.164:2181
... ..
[zk: 172.31.31.164:2181(CONNECTED) 0] ls /
[zookeeper]
[zk: 172.31.31.164:2181(CONNECTED) 1] create /zk "myData"
Created /zk
[zk: 172.31.31.164:2181(CONNECTED) 2] get /zk
myData
... ..
[zk: 172.31.31.164:2181(CONNECTED) 3] set /zk "newData"
... ..
[zk: 172.31.31.164:2181(CONNECTED) 4] get /zk
newData
... ..
[zk: 172.31.31.164:2181(CONNECTED) 5] delete /zk
```

上面的命令在 ZooKeeper 集群中创建了一个“/zk”键，并写入值 **myData**，读取确认后将此键内容改为 **newData**，再次确认无误，最后删除这个键。

4.2.2 部署 Mesos

在 Mesos 的早期版本里，部署 Mesos 服务需要从源代码进行编译。这个过程就像编译 Linux 内核，是一件费力耗时的事情，在 Mesos 的 Apache 项目网站^①至今还记录着这种从源代码进行编译的安装方式。直到 Mesos 0.19.1 版本以后，Mesos 所属的 Mesosphere 公司网站上开始提供 deb 和 rpm 格式的二进制安装包，这才将 Mesos 集群的部署变得容易起来。

现在依然可以在 <http://open.mesosphere.com/downloads/mesos/> 这个网址上直接下载这两种格式的安装程序，并依据自己的系统发行版进行安装。

Debian 和 Ubuntu 发行版可以选择 deb 格式的部署包，如下所示。

```
$ curl -Lo mesos.deb http://repos.mesosphere.com/ubuntu/pool/main/m/mesos/
mesos_1.1.0-2.0.107.ubuntu1604_amd64.deb
$ sudo apt update
$ sudo apt install gdebi-core
$ sudo gdebi mesos.deb
```

^① <http://mesos.apache.org/getting-started/>

这里使用了 `gdebi` 工具来在安装 `deb` 包的时候自动安装所需的依赖，最后这两个步骤也可以替换成 Ubuntu 内置的 `dpkg` 原生的安装方式，如下所示。

```
$ sudo dpkg -i mesos.deb
$ sudo apt -f install
```

这种方式在前一步虽然能够将程序安装上去，但会在最后的地方提示缺少依赖出错，然后再通过后一个命令进行修复。由于有错误发生，若想把安装过程自动化，就会不太友好，因此不推荐这样安装。

RHEL 和 CentOS 发行版可以选择 `rpm` 格式的部署包，如下所示。

```
$ curl -Lo mesos.rpm http://repos.mesosphere.com/el/7/x86_64/RPMS/
mesos-1.1.0-2.0.107.centos701406.x86_64.rpm
$ sudo yum --nogpgcheck localinstall mesos.rpm
```

在 RHEL 7 或 CentOS 7 之后的版本中，推荐用 `dnf` 替代 `yum` 工具，因此第二个命令也可以写成下面这样。

```
$ sudo dnf install mesos.rpm
```

在所有节点上依次完成 Mesos 服务的安装，就可以准备启动集群了。

4.2.3 启动 Master 节点

Mesos 安装完成后，会在 “`/usr/sbin`” 目录下生成许多可执行文件和脚本，如下所示。

```
$ ls /usr/sbin/mesos-*
mesos-agent
mesos-daemon.sh
mesos-master
mesos-agent
mesos-start-agents.sh
mesos-start-cluster.sh
mesos-start-masters.sh
mesos-start-slaves.sh
mesos-stop-agents.sh
mesos-stop-cluster.sh
mesos-stop-masters.sh
mesos-stop-slaves.sh
```

其中与 “slave” 相关的文件都是为了与 Mesos 1.0 版本以前的 Mesos 文件命名兼容而

保留的。现在“slave”的名称已经被“agent”所取代，而这些带“slave”的文件内容与相应的“agent”文件的完全相同，可以通过文件的 MD5 码证实这一点，如下所示。

```
$ md5sum /usr/sbin/mesos-agent
64487d55900c247736ddd9e2cd9ba876 /usr/sbin/mesos-agent
$ md5sum /usr/sbin/mesos-agent
64487d55900c247736ddd9e2cd9ba876 /usr/sbin/mesos-agent
```

在 Mesos 的安装包中，提供了一系列用于自动化部署配置脚本，也就是上面列出的“mesos-start-cluster.sh”“mesos-stop-cluster.sh”这些文件。不过，这些脚本依然需要用户在各个节点上分别提前准备配置文件，并添加与执行部署节点的 SSH 信任，实际上并不是十分好用，还会由于部署节点与其他节点之间的 SSH 免密码登录而增加安全隐患。另一方面，这些脚本隐藏了 Mesos 服务本来并不复杂的实际启动过程，对进一步了解 Mesos 也没什么好处。因此完全可以抛开这些脚本，直接使用 Mesos 的原始参数创建节点。

建议使用 Systemd 等服务管理工具，将 Mesos 注册成系统服务，然后通过系统服务的方式启动，如下所示。

```
$ cat << EOF | sudo tee /etc/systemd/system/mesos-master.service
[Unit]
Description=Mesos Master Service

[Service]
ExecStart=/usr/sbin/mesos-master \
    --ip=172.31.31.164 \
    --port=5050 \
    --advertise_ip=52.79.176.188 \
    --log_dir=/var/log/mesos \
    --work_dir=/var/run/mesos \
    --zk=zk://172.31.31.164:2181/mesos \
    --quorum=1
Restart=always

[Install]
WantedBy=multi-user.target
EOF

$ systemctl enable mesos-master
$ systemctl start mesos-master
```

其中--hostname 参数是节点对外访问的地址，譬如作为 Mesos Web 自动更新界面数据的服务回调地址，它应该配置成一个从服务的控制端（比如用户的笔记本电脑）上能够

访问到的地址或域名，如果用户是从外网访问的，则该地址需要配置成节点的外网 IP。而 `--ip` 参数是服务的监听地址，它应该配置成一个对其他的 Mesos 节点能够访问的地址或域名，通常使用所在局域网的内网 IP。

`--port` 参数指定 Mesos Master 提供服务的端口，默认就是 5050，可省略，这里只是为了让这个端口信息更明确才写上。`--log_dir` 和 `--work_dir` 参数分别指定了 Mesos 的日志和数据文件目录，其中后者是必须指定的，否则 Mesos 服务将无法启动。

`--zk` 和 `--quorum` 参数都是与 Master 节点的高可用特性相关的。`--zk` 参数指定用于存储 Leader 信息和协助选举的 ZooKeeper 服务地址和存储信息的键位置。若使用的 ZooKeeper 是一个集群，则应该依次列出每一个 ZooKeeper 实例的地址，使用逗号分隔开，例如 `"zk://172.31.31.164:2181, 172.31.27.243:2181, 172.31.27.16:2181/mesos"`。若使用的 ZooKeeper 配置了用户验证，就应该在地址前加上用户名和密码的明文信息，例如 `"zk://username: password@172.31.31.164:2181, 172.31.27.243:2181, 172.31.27.16:2181/mesos"`。`--quorum` 参数是 Paxos 协议中的 Leader 当选所需投票数，其取值应多于集群中一半 Master 节点的最小整数值。例如有 5 个节点的集群，半数就是 2.5，所以 Quorum 的取值应该为 3，即只要任何节点在投票时收到 3 个或以上的投票，即可将自己转换为 Leader。对于单个 Master 节点的情况，可将 Quorum 值设置为 1。

为了避免选举出现得票数平局的情况，Mesos Master 的个数通常应该是单数个。表 4-1 展示了 Master 节点个数和 Quorum 取值的关系。

表 4-1 Master 节点个数和 Quorum 的取值

Master 节点个数	1	3	5	7
Quorum 取值	1	2	3	4

如果要构建高可用的 Mesos Master 集群，只需相应调整 `--quorum` 参数的值，然后在每个节点上分别启动 Mesos Master 服务。这些服务会在集群有足够的 Mesos Master 实例启动后，自动通过 ZooKeeper 协商选出 Leader 节点。一旦 Leader 选举完成，Mesos Master 的服务便可以使用了。

此时，通过浏览器访问任意一个 Mesos Master 节点的 5050 端口，都会打开 Mesos 集群的管理界面，如图 4-4 所示。

这里有一个小技巧，Mesos 的管理界面代码文件默认被安装在 `"/usr/share/mesos/webui/master/static/"` 目录下，如果需要对界面内容进行定制，可以修改这个目录中相应文件的内容。

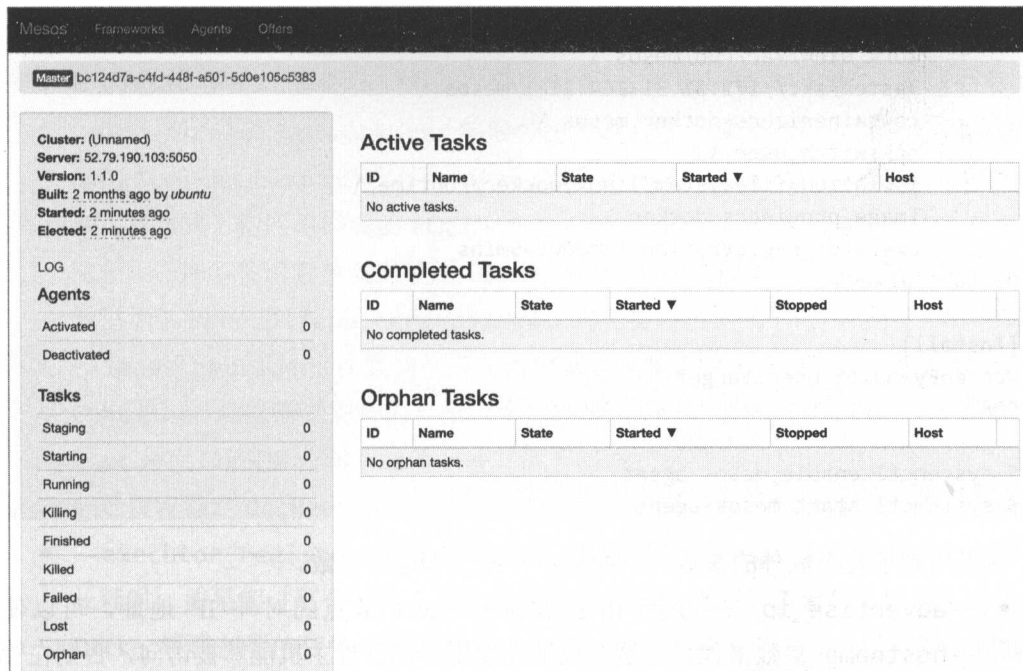


图 4-4 Mesos 的管理界面

4.2.4 添加 Agent 节点

Mesos Agent 节点的部署与 Master 节点差不多，只是不再需要进行 Leader 选举，因此也不用指定 `--zk` 和 `--quorum` 这样的参数。取而代之的是，需要使用 `--master` 参数告知 Mesos Agent 要连接的 Mesos Master 地址，它的值应该与 Mesos Master 所使用的 `--zk` 参数值相同。

同样建议使用 Systemd 等服务管理的工具将 Mesos Agent 作为系统服务进行管理，并以系统服务的方式启动，如下所示。

```
$ cat << EOF | sudo tee /etc/systemd/system/mesos-agent.service
[Unit]
Description=Mesos Agent Service

[Service]
ExecStart=/usr/sbin/mesos-agent \
  --ip=172.31.27.243 \
  --port=5051 \
  --advertise_ip=52.78.99.53 \
```



```
--log_dir=/var/log/mesos \  
--work_dir=/var/run/mesos \  
--master=zk://172.31.31.164:2181/mesos \  
--containerizers=docker,mesos \  
--no-switch_user \  
--isolation=filesystem/linux,docker/runtime \  
--image_providers=docker \  
--executor_registration_timeout=5mins  
Restart=always  
  
[Install]  
WantedBy=multi-user.target  
EOF  
  
$ systemctl enable mesos-agent  
$ systemctl start mesos-agent
```

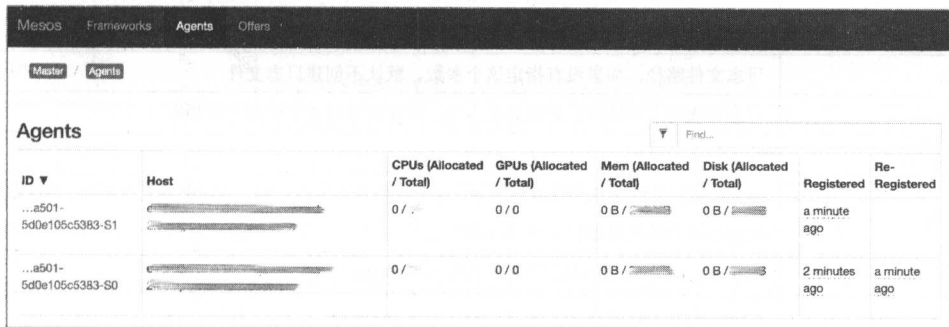
这里用到了几个额外的参数，下面重点介绍一下这些参数。

- **--advertise_ip** 参数用于指定 Mesos Agent 节点的对外 IP 地址，可以使用 **--hostname** 参数来代替，这个地址需要用户的管理主机能够被访问，否则在管理界面上进入 Agents 页时将无法动态更新状态。
- **--no-switch_user** 参数使得 Mesos 直接使用启动 Agent 服务的用户来执行所指定的任务内容，而不会尝试先切换为提交这个任务的用户。在使用 Unified Container 特性时，有些目录必须使用 Root 用户访问，这个参数可以避免 Mesos Agent 启动任务时切换为普通用户而使得任务执行失败。
- **--containerizers** 参数用于指定运行在这个 Mesos 集群中的框架能够使用的资源隔离（容器化）方式，默认为 **mesos**，指的是采用 Mesos 内置的经典容器化方式，它和其他容器一样使用基于内核 Namespace 的特性对资源访问进行控制。可选值是 **docker**，表示调用 Docker 服务来进行运行时资源的隔离。在 Mesos 1.0 版本以前，使用 Docker 容器化方式是唯一的利用容器镜像运行任务的方法，Mesos 在 1.0 版本以后推出了“Unified Container”功能，使得 Mesos 内置的容器化方式不依赖 Docker 服务也可以支持 Docker 镜像了。这里为了在之后的例子中对比演示两种容器化方式的差异，将值设置为了 **docker,mesos**，表示同时启用两种容器化方法。
- **--isolation** 参数仅仅对 Mesos 容器化的任务有效，它指定了当前节点上运行的任务使用什么方式进行资源配额的限制和管理，资源管理包括 CPU、内存、磁盘、文件系统、网络、数据卷等许多方面。这个参数的初始化过程比它看起来要复杂很多，

例子中设置的 `filesystem/linux,docker/runtime` 指定了文件系统和容器运行时的管理方式，实际上 Mesos 启动时还会经过一系列判断，然后自动补上 `volume/image`、`network/cni` 等额外的限制方法，具体过程可以参考 Mesos 源代码的“`containerizer.cpp`”文件^①。Mesos 默认的 `posix/cpu` 和 `posix/mem` 管理 CPU 和内存方式，指的是经典的 POSIX 方式（类似 `ulimit` 命令）来控制每个任务的资源使用，许多使用者更倾向使用 CGroup 来管理 CPU 和内存，为此也可以在这个参数里加上 `cgroups/cpu,cgroups/mem`，它会覆盖默认的 CPU 和内存管理策略。

- `--image_providers` 参数同样仅仅对 Mesos 容器化的任务有效，它指定 `Unitified Container` 功能需要支持的容器镜像种类列表，可选值包括 `docker` 和 `appc`。其中 `appc` 是由 CoreOS 公司主导的一种开放容器格式标准，主要用在 Rkt 等容器技术上，此处仅仅指定 `docker`。
- `--executor_registration_timeout` 参数配置了 Mesos 上任务启动时允许的最长耗时，如果超过这个时长，任务仍然没有启动完成，则认为任务出错。对于使用 Docker 镜像部署的任务，可能需要较长的时间来下载镜像，建议将此时间设置为 5 分钟或更长。

启动完成后，回到 Mesos 的管理界面，单击“Agents”标签页，可以看到所有已经加入到集群中的 Agent 状态，如图 4-5 所示。



Mesos Frameworks Agents Offers							
Master / Agents							
Agents							
ID ▼	Host	CPU (Allocated / Total)	GPU (Allocated / Total)	Mem (Allocated / Total)	Disk (Allocated / Total)	Registered	Re-Registered
...a501-5d0e105c5383-S1	[Host Icon]	0 / 0	0 / 0	0 B / 0 B	0 B / 0 B	a minute ago	
...a501-5d0e105c5383-S0	[Host Icon]	0 / 0	0 / 0	0 B / 0 B	0 B / 0 B	2 minutes ago	a minute ago

图 4-5 Mesos 的 Agent 节点管理界面

这个集群现在还不能做什么具体的事情，后面几节会介绍 Mesos 中比较常用的两个通用服务管理框架（Marathon 和 Chronos）以及如何运用这两个框架在 Mesos 上运行任意的自定义服务。

^① <https://github.com/apache/mesos/blob/master/src/slave/containerizer/mesos/containerizer.cpp>

4.2.5 Mesos 服务的启动参数

除了前面已经列出的参数以外，Mesos 服务还有许多可选的启动参数。以下列举其中比较常用的一些，完整的列表可以参考官方文档^①。

Mesos 服务的参数分为三种类型：通用项（Master 和 Agent 都支持）、只有 Master 支持的以及只有 Agent 支持的。

1. 常用的通用项

Mesos Master 和 Mesos Agent 皆可使用的启动参数及其说明如表 4-2 所示。

表 4-2 Mesos 的通用启动参数及其说明

参 数	说 明
--ip	服务监听的 IP 地址，一般使用节点的内网地址，这个地址需要能被集群中其他节点访问
--port	服务监听的端口，Master 默认是 5050，Agent 默认是 5051
--advertise_ip	Mesos 的 Web 界面向 Mesos 请求数据时使用的地址，需配置为访问端能够访问的地址，若 Mesos 部署在公有云，一般指定为节点的公网地址
--advertise_port	Mesos 的 Web 界面向 Mesos 请求数据时使用的端口，Master 默认是 5050，Agent 默认是 5051
--hostname	指定 Master 节点在 ZooKeeper 注册的名称，或是 Agent 节点向 Master 上报使用的名称，这里可写 IP 地址，Mesos 在使用时会自动查询转换成主机名称
--no-hostname_lookup	默认情况下 Mesos 会在注册 ZooKeeper 或上报数据前，将节点的 IP 地址转换为主机名称，这个参数会阻止这种转换，而直接使用 IP 地址进行注册和上报
--log_dir	日志文件路径，如果没有指定这个参数，默认不创建日志文件
--logbufsecs	日志每次写入本地前，先缓存多少秒，从而减少日志文件的写入频率
--logging_level	日志记录的级别
--work_dir	Mesos 服务的持久化文件存储目录，这个参数是必须指定的，否则服务无法启动，不宜使用/tmp 这种会被自动清理的位置作为存储目录
--modules	指定 Mesos 启动时加载额外模块的配置文件路径

这些参数都可被 Master 和 Agent 节点使用，其中的几个与地址和端口相关的参数比较容易混淆，配置时需要留意。一般来说，如果没有端口冲突，--port 和 --advertise_port 都可不用指定。--ip 参数在节点只有一个网卡的时候也不必指定，Mesos 会自动获取首个网卡的 IP 地址绑定。如果所有需要访问 Web 界面的用户都能直接访问 Mesos 绑定的 IP 地址，就不必指定 --advertise_ip，否则通常将它指定为用户能够访问的 IP 地址。

^① <http://mesos.apache.org/documentation/latest/configuration/>

`--hostname` 参数通常不必指定，有时在某些服务框架日志中出现 Mesos Master 注册地址与收到 Offer 的来源地址不一致的错误时，可以为 Master 节点指定此参数。

2. 常用的 Master 专属配置项

Mesos Master 专用的启动参数及其说明如表 4-3 所示。

表 4-3 Mesos 的 Master 专属启动参数及其说明

参 数	说 明
<code>--zk</code>	对于高可用集群来说，这个参数是必需的，表示使用的 Zookeeper 接口地址，支持多个地址，地址之间用逗号分隔
<code>--quorum</code>	必须指定的参数，表示 Leader 选举时所需的最小节点数目，值应该为大于 Master 节点总数一半的最小整数
<code>--cluster</code>	指定在 Web 界面上显示的集群名称
<code>--authenticate_agents</code>	仅允许授权通过的 Agent 节点注册
<code>--authenticate_frameworks</code>	仅允许授权通过的服务框架注册
<code>--authenticators</code>	设定认证方式，默认为 CRAM-MD5。指定其他认证方式时需要通过 <code>--modules</code> 加载相应模块
<code>--authorizers</code>	设定权限控制方式，默认为 local，即使用 ACL 文件配置。指定其他权限控制方式时需要通过 <code>--modules</code> 加载相应模块
<code>--credentials</code>	指定存放身份凭证的配置文件
<code>--acls</code>	指定 ACL 访问控制规则的配置文件路径
<code>--whitelist</code>	指定能够调度任务的 Agent 列表，通常在进行 Agent 节点组件升级时会需要这个功能，如果未指定这个参数，默认所有节点都可以调度
<code>--agent_removal_rate_limit</code>	Agent 健康检查失败时被移除的速率上限，是个数与时间的比值，例如“2/10mins”代表每十分钟最多允许移除两个 Agent 节点

Mesos Master 的认证功能主要是为了确保只有受信任的 Agent 和服务框架可以与 Mesos 集群进行交互。Mesos 在其 0.15.0 版本开始支持服务框架认证，使用 `--authenticate_frameworks` 参数开启。在其 0.19.0 版本开始支持 Agent 节点认证，使用 `--authenticate_agents` 参数开启。其认证功能基于 Cyrus SASL 库，支持多种认证机制，默认使用 CRAM-MD5 认证，可使用 `--authenticators` 参数修改。

CRAM-MD5 使用 `principal` 和 `secret` 值对作为认证凭证，`principal` 用于表示验证者的身份。在 Master 节点启动时，`--credentials` 参数指定一个所有可用身份的列表文件，内容如下所示。

```
{
```

```
"credentials": [  
  {  
    "principal": "user01",  
    "secret": "password"  
  },  
  {  
    "principal": "user02",  
    "secret": "pa55word"  
  }  
]
```

假如这个文件存放在“/mesos/master/credentials”位置，则 Master 启动的参数可以写为 `--credentials=file:///mesos/master/credentials`。在启动 Agent 时，需要用 `--credential` 参数指定一个身份验证文件，内容如下所示。

```
{  
  "principal": "user01",  
  "secret": "password"  
}
```

假如这个文件存放在“/mesos/agent/credential”位置，则 Agent 启动的参数可以写为 `--credential=file:///mesos/agent/credential`。

服务框架的验证通常也在框架启动时指定，例如 Marathon 框架需要给予 `--mesos_authentication` 参数开启认证功能，用 `--mesos_authentication_principal` 参数指定使用的 principal 名称，然后将 secret 内容存放到一个文件中，用 `--mesos_authentication_secret_file` 参数指定该文件路径。

当使用了身份验证后，在 Mesos 中还可以根据用户身份进行更详细的访问权限控制。在默认方式下，Mesos 采用 ACL 配置文件来设定需要控制的内容。ACL 是通过访问控制列表（Access Control Lists）的简称。在 Mesos 中，它是一种 Json 格式的配置文件，内容格式如下所示。

```
{  
  <"权限控制点">: [  
    {  
      "principals": { <"values"或"type">: ["用户"] },  
      "<选项>": { "<values 或 type>": ["<条件>"] }  
    },  
    { ... }  
  ],  
}
```



```
<"另一个权限控制点">: [ ... ]
}
```

其中的“权限控制点”是 Mesos 允许进行权限控制的功能点，在 v1.1 版本中一共有 17 个可控制功能点^①，包括限制服务框架可用角色的 `register_frameworks`、限制服务框架可用用户的 `run_tasks` 等。`principals` 指定认证者的身份，根据配置的上下文，它既可以表示一个登录的用户，也可以表示注册的一个服务框架。它的内容可以是 `type` 或 `values`，当使用 `type` 时只能表示两个特殊值：表示“任意值皆可”的 `ANY` 和表示“所有值皆不可用”的 `NONE`。当使用 `values` 时，需要提供一个具体的可用值列表。“选项”和“条件”的内容与所属的“权限控制点”有关。这样描述起来有点抽象，下面举一个配置的示例。

```
{
  "permissive" : true,
  "register_frameworks": [
    {
      "principals": { "values": ["marathon"] },
      "roles": { "values": ["prod", "uat", "dev"] }
    },
    {
      "principals": { "values": ["chronos"] },
      "roles": { "values": ["*"] }
    },
    {
      "principals": { "values": ["marathon", "chronos"] },
      "roles": { "type": "NONE" }
    }
  ]
}
```

这个配置表示，使用 `marathon` 身份登录的服务框架能且只能使用 `prod`、`uat` 或 `dev` 这三个角色的资源。而使用 `chronos` 身份登录的服务框架，只能使用默认的“*”角色。开头的 `permissive` 为权限设置一个默认值，将它设置为 `true` 或不设置时，表示“若无明确禁止，则全部允许访问”，将它设置为 `false`，表示“若无明确允许，则全部禁止访问”。

准确来说，`--acls` 参数也可以用在 Mesos Agent 服务上，不过实际上很少需要这样做，因此本书姑且将此参数划归到 Master 专属的常用参数范畴。Mesos 支持多种权限控制的机制，若希望使用 ACL 配置文件以外的控制方式，可加载相应模块，然后使用 `authorizers`

^① <http://mesos.apache.org/documentation/latest/authorization/>

参数修改。

此外，Mesos 提供 `--agent_removal_rate_limit` 参数的目的在于避免因偶发的短时间网络故障使得大量 Agent 由于心跳包未送达，而导致整个集群瞬间崩溃的后果。通过限制单位时间里移除 Agent 节点数量，确保在 Master 节点和 Agent 节点的通信重新恢复之前，集群能够维持一段时间，同时在集群真的出现永久性故障时，所有 Agent 节点最终也能被全部移除。

3. 常用的 Agent 专属配置项

Mesos Agent 专用的启动参数及其说明如表 4-4 所示。

表 4-4 Mesos 的 Agent 专属启动参数及其说明

参 数	说 明
<code>--master</code>	Master 节点所在地址，通常使用 ZooKeeper 集群的地址和路径表示
<code>--attributes</code>	赋予机器的键值对属性，格式键与值之间使用冒号分隔，多个属性之间使用分号分隔，例如 <code>rack:2;unit:u1</code>
<code>--authenticatee</code>	设定认证方式，默认为 CRAM-MD5
<code>--credential</code>	身份凭证配置文件，和 Master 节点的 <code>--credentials</code> 类似，不同的是，Agent 节点的配置文件里只能有一个身份配置
<code>--cgroups_limit_swap</code>	需要同时限制内存和 swap 的用量，默认情况下只限制内存
<code>--containerizers</code>	指定启用的容器化实现机制，包括 <code>mesos</code> 、 <code>docker</code> 和 <code>external</code> ，默认只启用 <code>mesos</code> 容器，可同时启用多个，使用逗号分隔
<code>--no-switch_user</code>	使用启动 Agent 服务的用户身份，而不是提交任务的用户身份来运行任务
<code>--isolation</code>	启用的隔离机制，该参数在本章的第 4.2.4 小节中已经介绍过
<code>--image_providers</code>	需要启用的 Unified Containers 镜像类型，可指定多个，使用逗号分隔，例如 <code>docker</code> 、 <code>appc</code>
<code>--executor_registration_timeout</code>	容器启动的超时时间，默认 1 分钟
<code>--default_role</code>	资源缺省分配的角色
<code>--resources</code>	指定当前节点所有可用资源的量

Attributes 是赋予特定 Agent 节点的任意键值对，在节点启动时通过 `--attributes` 参数赋予，这个键值对在 Mesos Master 向服务框架提供 Offer 时会一起带上，这样自定义的服务框架可以通过节点的属性来进行选择性的资源过滤。

Role 是 Mesos 中实现租户隔离的方式。每个 Role 相当于一个独立的租户，拥有一个全局唯一的名称和属于该 Role 的专用计算资源，Agent 启动时可以使用 `--resources` 参数为特定的 Role 预留资源。对于没有指定 Role 的资源，默认情况下属于名称为 “*” 的特殊 Role，用户也可以使用 `--default_role` 参数来将当前 Agent 所有未指定 Role 的资源赋予

特定的一个 Role。

Mesos 为 Role 预留资源的作用主要是为了确保一些优先级很高的服务框架和任务总是能够获得必要的资源并马上运行，而不用等待其他的服务框架释放资源。每个服务框架运行时都属于特定 Role 的租户空间，并消费属于这个 Role 的计算资源。一般在框架启动时可以选择 Role，例如 Marathon 使用 `--mesos_role` 参数指定自己所属的 Role 名称，若未指定则默认属于 “*” 这个 Role。在同一个 Role 中的所有服务框架所使用的资源总量不能超过此 Role 拥有的最大值。

在 Mesos Agent 启动后，会尝试自动识别可用的 CPU、内存和磁盘等资源，并把这些资源全部赋予该节点默认的 Role。有两种方式可以对资源的归属进行定制：一种是静态指定方法，也就是在启动 Agent 服务时，使用 `--resources` 参数指定；另一个种是动态指定方法，需要在节点加入集群后，通过 Mesos 的 API 调用进行事后修改。由于每个节点的可用计算资源量通常不会改变，因此静态指定的方式目前依然比较流行。`--resources` 参数的值可以用分号分隔的资源列表或 Json 格式的资源分配配置，或者包含资源分配配置内容的 Json 文件目录。

分号分隔表示法是比较简单的一种表示方式，将资源类型和数量用冒号分隔开，直接使用分号分开不同的资源，最后对于需要指定 Role 的资源，在小括号中注明所属的 Role 名称。若未指定 Role，则属于该节点的默认 Role（一般是 “*” 这个特殊的 Role，可使用 `--default_role` 参数修改），如下所示。

```
/usr/sbin/mesos-agent \  
  --resources="cpus:4;gpus:4;mem:2048;disk:409600; \  
  cpus(demo):8;mem(demo):4096;disk:409600(demo)" \  
  ...
```

那么这个节点一共具有 12 核 CPU 资源、4 核 GPU 资源、6144MB 的内存和 800GB 的磁盘资源。将其中 8 核 CPU、4096MB 内存和 400GB 的磁盘预留给名称为 “demo” 的 Role，其他的属于默认 Role（即 “*” 这个 Role）。也可以明确地将资源指定给 “*” 这个 Role，如下所示。

```
/usr/sbin/mesos-agent \  
  --resources="cpus(*):4;gpus(*):4;mem(*):2048;disk(*):409600;
```

值得一提的是，除了这两种基于 Agent 节点的资源预留方法，Mesos 还提供了一种在整个集群中为特定 Role 预留资源的机制，Quota。它的使用方法与节点资源的动态分配比较相似，同样需要使用 Mesos API 创建，关于 Quota 的详细用法这里就不再展开，有兴趣

的读者可参考相应文档^①。

4.3 使用 Marathon 管理服务

4.3.1 部署 Marathon

Marathon 是由 Mesosphere 公司开发的 Mesos 后台任务管理框架，于 2013 年开源。它相当于操作系统中的 Init.d 或是 Supervisord 这类管理长时间运行任务的服务，比如 Web 应用等。此外，它会在任务异常退出时，自动重新启动任务实例以确保任务的状态正常。例如，当一个群集中的节点在半夜出现故障时，Marathon 会自动在另一个可用的节点里重新调度出现故障的任务或应用服务。

作为一个服务框架，Marathon 同样包含调度器（Scheduler）和执行器（Executor）两个部分。在部署 Marathon 时，实际上部署的是它的调度器服务。而执行器是在任务启动时，由 Agent 节点自动运行的。

首先从 Marathon 的 GitHub 页面^②下载它的最新发行版本，并解压到系统目录中，如下所示。

```
$ curl -Lo marathon.tgz http://downloads.mesosphere.com/marathon/v1.3.5/marathon-1.3.5.tgz
$ tar xzf marathon.tgz
$ sudo mv marathon-1.3.5 /usr/local/marathon
```

Marathon 服务默认支持高可用部署，它和 Mesos 同样地使用 ZooKeeper 进行 Leader 选举并保存集群 Leader 信息。因此，在启动时需要通过 `--zk` 参数告诉 Marathon 使用的 ZooKeeper 服务地址和路径。多个使用相同 ZooKeeper 地址和路径的 Marathon 会自动组成高可用的 Marathon 服务集群。

通常建议将 Marathon 服务部署在独立于 Mesos 的单独节点上，Marathon 发行包中的“bin/start”文件是启动服务的入口。需要注意的是，Marathon 服务需要“libmesos.so”这个文件，可以直接在这个节点上安装 Mesos 的二进制包，或是从安装了 Mesos 的节点拷贝

① <http://mesos.apache.org/documentation/quota/>

② <https://github.com/mesosphere/marathon>

“/usr/lib/libmesos.so”文件，放到该节点的“/usr/lib”或“/usr/local/lib”目录。

建议同样通过 Systemd 来管理 Marathon，如下所示。

```
$ cat << EOF | sudo tee /etc/systemd/system/marathon.service
[Unit]
Description=Mesos Marathon Service

[Service]
ExecStart=/usr/local/marathon/bin/start \
  --master zk://172.31.31.164:2181/mesos \
  --zk zk://172.31.31.164:2181/marathon
Restart=always

[Install]
WantedBy=multi-user.target
EOF

$ systemctl enable marathon
$ systemctl start marathon
```

其中 `--master` 参数可以直接指定 Mesos Master 节点的 IP 列表，也可以使用 Mesos 的 ZooKeeper 地址和路径。此外，Marathon 服务还有许多可选的启动参数，例如指定框架使用名称为“marathon”的 Role，并开启框架身份验证，如下所示。

```
/usr/local/marathon/bin/start \
  --mesos_role marathon \
  --mesos_authentication \
  --mesos_authentication_principal marathon \
  --mesos_authentication_secret_file /mesos/marathon.secret \
  ...
```

默认情况下，Marathon 启动应用的超时时间是 5 分钟，对大多数场景来说已经足够。如果应用需要下载容器镜像，且速度实在过于缓慢，可以通过 `--task_launch_timeout` 参数延长这个时间，单位是 ms。例如将启动超时时间配置为 10 分钟，如下所示。

```
/usr/local/marathon/bin/start \
  --task_launch_timeout=600000
...
```

Marathon 服务有十分漂亮的 Web 管理界面，默认监听 Marathon 节点的 8080 端口，如图 4-6 所示。

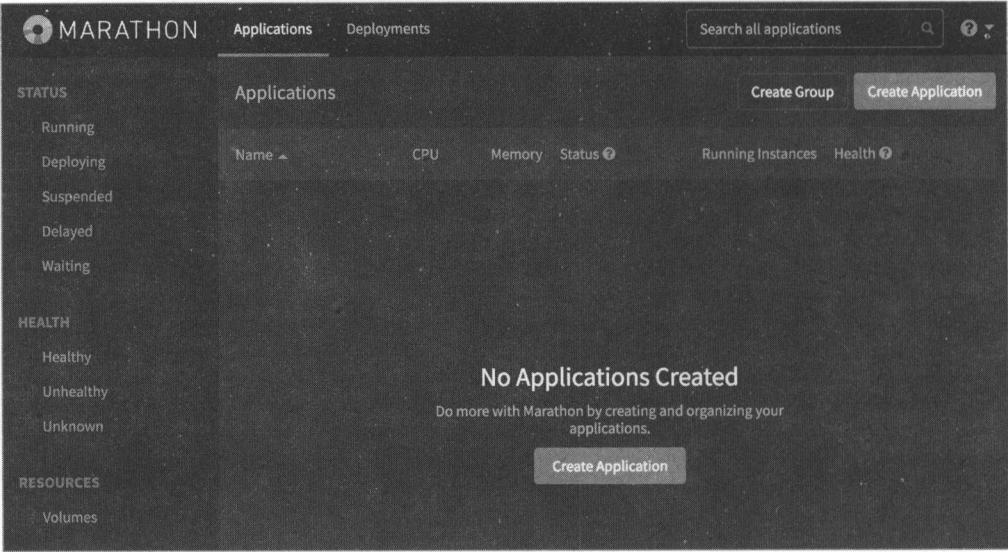


图 4-6 Marathon 的 Web 管理界面

4.3.2 添加一个应用

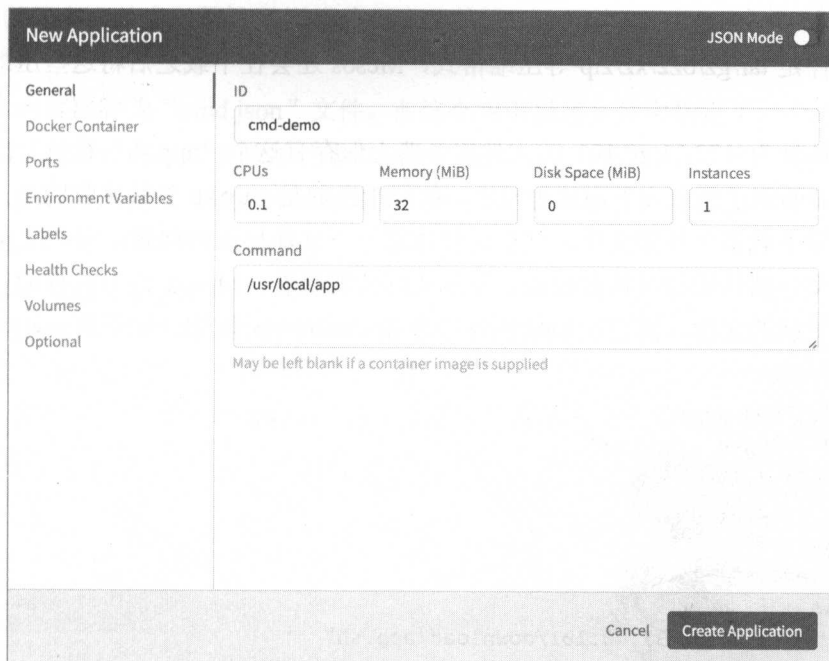
Mesos 对调度单元的定义是任务（Task），这是一个与资源使用相关的细粒度概念。Marathon 提出了几个更易于用户使用的概念，最核心的概念是应用（Application）以及应用组（Group）。应用是一组相同任务的集合，而每一个任务在 Marathon 中成为应用的实例（Instance），使用者可以按需扩缩实例的个数，每个实例会被 Mesos 调度到不同的 Agent 上运行。应用组则是多个相关联的不同应用的集合，例如一个系统中的许多微服务，它们直接存在相互的调用和依赖，共同构成一个有业务意义的部署实体。

Marathon 支持在 Mesos 集群上启动基于 POSIX/CGroup 的 Mesos 容器和现代的 Docker 容器，并提供方便易用的 Web UI 和功能丰富的 Restful API 用于创建、修改和删除应用，以及查询有关运行实例的信息。单击右下角蓝色的“Create Application”按钮就能通过 Web UI 创建 Marathon 应用，然后在弹出的窗口填写相关的属性信息，如图 4-7 所示。

界面的右上角有一个“JSON Mode”按钮，使用它可以切换简易模式和 Json 模式两种配置方法，如图 4-8 所示。

简易模式使用起来比较方便，但功能不如 Json 模式完整。例如在当前版本的 Marathon 中，简易模式还无法配置 Mesos 容器使用的镜像（即 Unified Container 特性）。通常可以先在简易模式中填充必要的参数，然后在 Json 模式中确认和补充，最后单击“Create

Application”按钮创建应用。



The image shows the 'New Application' form in the Marathon Web UI, set to 'General' mode. The form has a sidebar on the left with tabs: General, Docker Container, Ports, Environment Variables, Labels, Health Checks, Volumes, and Optional. The 'General' tab is active. The form fields are as follows:

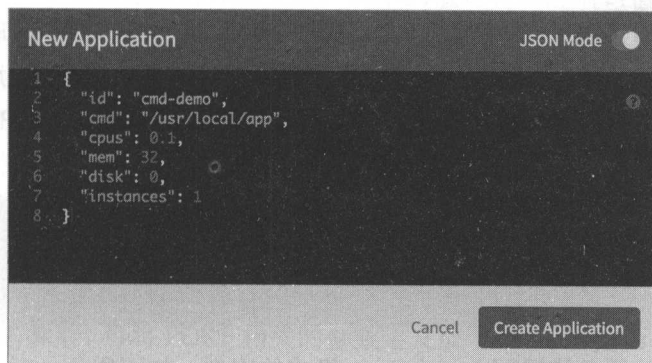
ID	CPU	Memory (MiB)	Disk Space (MiB)	Instances
cmd-demo	0.1	32	0	1

Command: /usr/local/app

May be left blank if a container image is supplied

Buttons: Cancel, Create Application

图 4-7 通过 Marathon Web UI 创建应用



The image shows the 'New Application' form in the Marathon Web UI, set to 'JSON Mode'. The form displays a JSON configuration for the application:

```
1 {  
2   "id": "cmd-demo",  
3   "cmd": "/usr/local/app",  
4   "cpus": 0.1,  
5   "mem": 32,  
6   "disk": 0,  
7   "instances": 1  
8 }
```

Buttons: Cancel, Create Application

图 4-8 Marathon 创建应用的 Json Mode

值得指出的是，Mesos 默认采用的 Mesos 容器只提供 CPU、GPU、内存、磁盘等方面的隔离，而应用运行的文件系统是与节点共用的，并未做隔离。这是因为 Mesos 最初被设计主要用于计算密集型的应用，这类应用的存储主要使用内存、分布式存储和本地的临时工作目录。因此，如果像上例那样指定“/usr/local/app”程序的绝对路径，则要求所有运行任务的 Agent 节点上都预先准备这个可执行文件，否则应用启动可能会失败，这与分布式

应用部署的初衷是相悖的。为此，Marathon 提供了一个 `uris` 属性来让应用在启动时能够通过 HTTP、FTP、HDFS、S3 等协议，从外部自动获取运行所需的文件到应用的运行目录。若下载的文件是 `tar/gz/bz2/xz/zip` 等压缩格式，Mesos 还会在下载之后将这些压缩包自动解压缩。

假设在地址为 `172.31.40.101` 的文件服务器上保存有这个“`app.sh`”待执行或部署的文件，可将前面的例子修改为下面这种写法。值得指出的是，`urls` 参数接收的是一个文件列表，支持批量下载多个外部文件。这里需要注意一下文件权限的问题，有些文章说 Mesos 会为下载的文件默认添加可执行权限，但从实际情况来看，Mesos 没有这样的行为，因此如果要执行下载的文件，需要在 `cmd` 属性中添加 `chmod +x` 操作，如下所示。

```
{
  "id": "cmd-demo",
  "cmd": "bash ./app.sh",
  "cpus": 0.1,
  "mem": 32,
  "disk": 0,
  "instances": 1,
  "uris": [
    "http://172.31.40.101/download/app.sh"
  ]
}
```

除了通过界面创建服务，许多运维人员更喜欢直接通过 Marathon 的 Restful API 来管理应用。为了让演示更直观，这个例子使用一个不需要本地文件的脚本作为应用的运行内容，脚本中使用 `nc` 命令监听节点的 1234 端口，并对所有访问它的 HTTP 请求返回字符串“ABCD”，如下所示。

```
while [ true ]; do
  echo 'HTTP/1.0 200 OK\r\nContent-Length: 7\r\n\r\nABCD' | \
  nc -l -p 1234;
done
```

首先将要创建的服务描述写入一个 Json 格式的文件，如下所示。

```
$ cat << EOF > marathon-cmd.json
{
  "id": "marathon-cmd-demo",
  "cmd": "while [ true ]; do echo 'HTTP/1.0 200 OK\r\nContent-Length: 7\r\n\r\nABCDEFG' | nc -l -p \${PORT0}; done",
  "cpus": 0.1,
  "mem": 32,
  "disk": 0,
```

```
"instances": 1,
"ports": [ 0 ]
}
EOF
```

将以上内容保存为“cmd.json”文件。在这个应用描述文件中指定了一个 `ports` 属性，这个属性用于列出该应用需要使用的端口列表，数值 0 表示使用一个随机端口。默认配置下所有的 Agent 节点都只开放 31000~32000 之间的端口，可以在启动 Mesos Agent 服务时通过 `--resources` 参数指定，如下所示。

```
/usr/sbin/mesos-agent \
--resources="ports(*):[7000-8000, 31000-32000]"
...
```

`ports` 属性实际上是 `portDefinitions` 属性的简便写法，更完整的写法如下所示。

```
"portDefinitions": [
{
  "port": 0,
  "protocol": "tcp"
}
]
```

在“ports”或“portDefinitions”列表中的每一个端口会依次在应用的运行环境中生成 `PORT0`、`PORT1`……等数值递增的环境变量，其中的值是相应的端口值。在这个例子中，将端口值指定为 0，则 Mesos 会在服务调度到的 Agent 节点上随机分配一个可用端口，并赋值给 `PORT0` 环境变量。在应用启动时，使用了 `$PORT0` 来获取变量值，让应用的实例监听这个分配的端口。

此外，`cpus`、`mem`、`disk` 属性是对应用所需资源的描述，`instances` 属性用于指定应用所需的实例数目，Marathon 会从 Mesos 提供的资源 Offer 中选择足够数量的匹配 Offer，并应用在相应的 Mesos Agent 节点上启动任务。

Marathon 的创建应用 API 路径是“/v2/apps”，需使用 POST 方式调用。为了避免反复配置 `curl` 命令的请求参数，可将 POST 相关的参数定义成一个 Shell 的命令别名，如下所示。

```
$ alias curl-post='curl -X POST -H "Content-type: application/json"'
```

假设 172.31.38.204 是其中一个 Marathon 节点，用以下命令从“cmd.json”文件中读取内容并发送给 Marathon。应用创建成功后会返回 Json 格式的应用信息。可使用 `jq` 命令对输出内容进行格式化，关于 `jq` 命令的使用可以查看其官方网站^①，如下所示。

① <https://stedolan.github.io/jq>

```
$ curl-post -Ls http://172.31.38.204:8080/v2/apps -d @marathon-cmd.json | jq '.'
{
  "id": "/marathon-cmd-demo",
  "cmd": "...",
  "args": null,
  "user": null,
  "env": {},
  "instances": 1,
  ... ..
}
```

通过 API 创建的应用同样会出现在 Web 界面中，如图 4-9 所示。

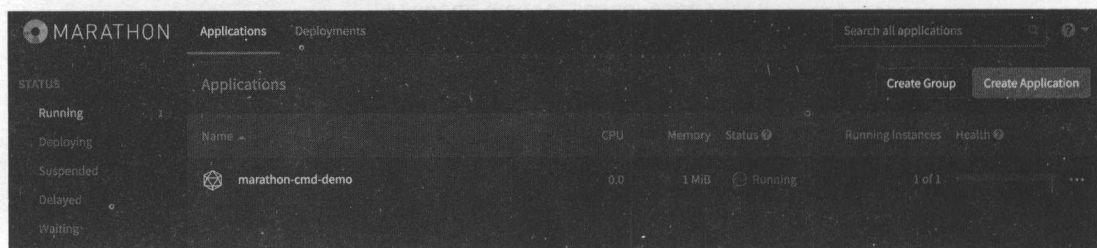


图 4-9 在 Marathon 中部署应用

此时通过 GET 请求访问 Marathon 服务的 “/v2/apps/<应用 ID>” 路径可以查询应用的详细信息，返回内容同样是 Json 格式，可以使用 jq 命令过滤返回的信息，例如应用部署的实际节点 IP 和监听的端口。

```
$ curl -s http://172.31.38.204:8080/v2/apps/marathon-cmd-demo | \
jq '.app.tasks[0].host'
172.31.27.243
$ curl -s http://172.31.38.204:8080/v2/apps/marathon-cmd-demo | \
jq '.app.tasks[0].ports[0]'
31238
```

访问这个服务被调度到的目标地址和端口，会返回字符串 ABCD，如下所示。

```
$ curl 172.31.27.243:31238
ABCD
```

使用 DELETE 方式访问 Marathon 服务的 “/v2/apps/<应用 ID>” 路径将删除指定的应用，如下所示。

```
$ curl -sX DELETE http://172.31.38.204:8080/v2/apps/marathon-cmd-demo | jq '.'
{
  "version": "...",
  "deploymentId": "259bc827-5f90-470b-8892-b928ebcf3af3"
}
```


4.3.3 使用 DC/OS 命令行工具

由于 DC/OS 系统内置了 Marathon 服务框架，它提供的命令行工具 `dcos` 可以直接用于操作和管理 Marathon 的应用，且比使用 `curl` 命令更加直观。因此这里不妨提前介绍一下这个工具的使用。

下载最新版本的 `dcos` 命令工具，并拷贝到 `PATH` 环境变量中的目录。然后就可以开始使用这个命令了，如下所示。

```
$ curl -O https://downloads.dcos.io/binaries/cli/linux/x86-64/dcos-1.8/dcos
$ chmod +x dcos
$ sudo mv dcos /usr/local/bin/
```

首先使用 `dcos config set` 设定要操作的目标 Marathon 服务地址，这里不能使用 Marathon 注册的 ZooKeeper 地址来代替。如果使用了高可用部署，只需指定其中任意一个 Marathon 节点的地址，如下所示。

```
$ dcos config set marathon.url http://172.31.38.204:8080
```

然后执行 `dcos marathon app` 命令就可以完成应用的创建、查看和删除操作了。以下三个操作的效果与前一小节相应的 `curl` 命令是等效的，但看起来就清爽许多。

```
$ dcos marathon app add marathon-cmd.json
$ dcos marathon app show marathon-cmd-demo
$ dcos marathon app remove marathon-cmd-demo
```

此外，`dcos` 命令也能够方便地调用其他 Marathon API，例如动态地修改服务参数，将应用的实例副本修改成两个，如下所示。

```
$ dcos marathon app update marathon-docker-demo instances=2
```

除了操作 Marathon，`dcos` 命令还有许多与 DC/OS 数据中心操作系统相关的功能，这些内容在第 4.6 一节中会详细说明。

4.3.4 使用 Docker 容器

Mesos 能够利用 Agent 节点的 Docker 服务来提供任务运行环境，由于集群的计算资源是由 Mesos 内核封装提供的，所有基于 Mesos 的服务框架都能直接获得运行 Docker 的能力。在介绍 Mesos Agent 服务启动参数时，已经说过 `--containerizers=docker`，`mesos` 参数表示同时开启 Docker 和 Mesos 两种容器化类型，此参数对于这个小节的内容来说是必

需的，否则在应用部署时会提示 Docker 容器类型不存在的错误。同时，这种运行方式要求集群中的 Agent 节点预先安装 Docker 服务。

在 Marathon 的应用描述中，使用 `container.type` 属性表示应用运行使用的隔离容器类型。当类型为“DOCKER”时，使用 `container.docker` 属性描述容器的细节参数，如下所示。

```
$ cat << EOF > marathon-docker.json
{
  "id": "marathon-docker-demo",
  "cmd": "python3 -m http.server 8080",
  "cpus": 0.5,
  "mem": 64.0,
  "instances": 1,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "python:3.5.1-alpine",
      "network": "BRIDGE",
      "portMappings": [
        {
          "containerPort": 8080,
          "hostPort": 31080,
          "protocol": "tcp"
        }
      ]
    }
  }
}
EOF
```

这个应用描述文件指定了使用 `python:3.5.1-alpine` 的 Docker 容器，在容器中执行 `python3 -m http.server 8080` 命令，并指定了将容器的 8080 端口映射到运行主机的 31080 端口。注意，这里的 `hostPort` 可用范围与第 4.3.2 小节介绍过的 `ports` 属性相同，默认情况下所有 Agent 都只开放 31000~32000 之间的 TCP 端口。如果要指定使用其他的端口，需要至少有一部分 Agent 节点启动时使用 `--resources` 参数开放相应的端口范围。创建这个应用，如下所示。

```
$ dcos marathon app add marathon-docker.json
```

查看应用的属性，并用 `jq` 命令过滤出应用示例运行的主机和端口，如下所示。

```
$ dcos marathon app show marathon-docker-demo | \
jq .tasks[0].host
172.31.27.16
$ dcos marathon app show marathon-docker-demo | \
jq .tasks[0].ports[0]
31080
```

访问这个地址会看到一个文件列表的 Web 页面（也可直接通过浏览器打开），如下所示。

```
$ curl 172.31.27.16:31080
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
...
</body>
</html>
```

此时，登录到相应的节点上，可以看到有一个 python 容器在运行，证实应用的确是由 Docker 服务运行的，如下所示。

```
$ docker ps
CONTAINER ID   IMAGE          COMMAND          ...
e18fe1143e53   python:...    "python3 ..."  ...
```

为了对比这种运行方式和 Unified Container 启动的应用的差异，先删除这个应用，如下所示。

```
$ dcos marathon app remove marathon-docker-demo
```

4.3.5 使用 Unified Container

Unified Container 是 Mesos 1.0 版本新增的特性，它增强了 Mesos 容器的功能，使之能够获取和解析 Docker、AppC 等格式的镜像文件，并直接通过这些镜像来创建应用。它需要 Mesos Agent 启动时包含 `--isolation=docker/runtime` 参数声明，且使用 `--image_providers` 参数列出需要开启的镜像格式支持。

以使用 Docker 镜像为例，首先在 `container.type` 属性指明使用容器类型为“MESOS”，然后依然使用 `container.docker` 属性设置镜像等与容器相关的参数。需要注

意的区别是，Mesos 容器不支持 Docker 的桥接网卡和端口映射的工作方式，因此与网络相关的配置应该用标准的 `ports` 属性指定。例如这个应用描述：

```
$ cat << EOF > marathon-unified.json
{
  "id": "marathon-unified-demo",
  "cmd": "python3 -m http.server \${PORT0}",
  "cpus": 0.5,
  "mem": 64.0,
  "instances": 1,
  "container": {
    "type": "MESOS",
    "docker": {
      "image": "python:3.5.1-alpine"
    }
  },
  "ports": [ 31081 ],
  "requirePorts": true
}
EOF
```

为了让测试结果与预期更加一致，建议使用 `requirePorts` 属性。在默认情况下，Marathon 在检查所获得的 Offer 时不会检查资源相应节点的端口使用情况，万一选择的节点不能提供服务所要求的端口，则会随机重选一个端口给应用使用。而 `requirePorts: true` 参数要求 Marathon 必须将应用的实例调度到所有指定端口均可用的节点上，以确保服务能够使用指定的端口号，否则将拒绝使用该资源 Offer。

创建这个应用，然后查看服务的运行状态，假设这个应用的副本被调度到了与前一个例子相同的节点上，如下所示。

```
$ dcos marathon app add marathon-unified.json
$ dcos marathon app show marathon-unified-demo | \
  jq .tasks[0].host
172.31.27.16
$ dcos marathon app show marathon-unified-demo | \
  jq .tasks[0].ports[0]
31080
```

访问这个地址同样会看到一个文件列表的 Web 页面，如下所示。

```
$ curl 172.31.27.16:31080
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```



```
<html>
...
</html>
```

再次登录到这个节点，检查节点上运行的 Docker 容器，这次无法找到相应的容器运行记录，如下所示。

```
$ docker ps
CONTAINER ID   IMAGE      COMMAND      ...
```

这证实了使用 Mesos 容器的 Unified Container 特性运行 Docker 镜像是不依赖于 Docker 服务的，因此也不再需要 Agent 节点预装 Docker 了。

值得一说的是，比较细心的读者可能会发现，使用 Mesos 容器启动的服务看到的文件列表内容与使用 Docker 容器启动同一个镜像时看到的不太一样，并不是在镜像的根目录，而是在只有两个日志文件（stderr 和 stdout）的目录中。这与 Mesos 的沙箱特性有关，Mesos Agent 在执行任务时，会先自动为每一个任务创建一个沙箱目录，用来作为这个任务的执行环境和存放相关的日志文件。这个沙箱目录存在于每个 Mesos Agent 启动时的 `--work_dir` 参数指定的地方，在之前介绍过使用 `uris` 属性为运行上下文添加的文件也都会被下载到这里。在本书中使用的“Mesos 容器”概念，在许多地方也被称为“Mesos 沙箱”。对于使用 Unified Container 特性运行镜像时，Mesos 依然会为任务创建一个沙箱目录，然后将沙箱目录自动挂载到容器运行环境中，并将该目录作为容器的工作目录。默认的挂载目录是 `/mnt/mesos/sandbox`，可在 Mesos Agent 启动时使用 `--sandbox_directory` 参数修改。

此外，Unified Container 的另一个特殊地方是，若需要从带账号的私有仓库下载镜像，由于不能依赖 Docker 服务的登录功能，它专门支持了一个额外的 `credential` 属性。假设 App 镜像来自一个需要登录的私有仓库，则写法如下所示。

```
"container": {
  "type": "MESOS",
  "docker": {
    "image": "my-private-repo.com/app",
    "credential": {
      "principal": "<账号>",
      "secret": "<密码>"
    }
  }
}
```

相比使用 Docker 的容器化方法时，需要登录到各个 Agent 节点去执行 `docker login` 命令，这算得上是 Unified Container 的一个隐含的好处了。

4.3.6 持久化卷存储

在非容器化的 Marathon 任务中，由于仅仅存在 CPU、内存的隔离，虽然通常建议每个应用仅在自己的沙盒目录中创建文件，但每个应用其实是可以使用绝对路径直接访问所在 Agent 节点上任何（权限允许的）目录和文件的。当应用升级时，必要的數據可以不用随着应用副本的重新部署而丢失。

而当 Marathon 引入了 Docker 容器以及 Unified Container 特性后，镜像方式运行的应用副本会在动态扩缩、应用升级等操作时，丢掉其中存储的所有数据。因此，Marathon 中也支持类似 Docker 数据卷挂载的能力，支持在容器中直接挂载 Agent 本地目录和基于 AWS 的云端存储磁盘，并通过 Flock 等第三方工具进行扩展。在容器的 `volumes` 属性中可以添加挂载卷的配置，其中包含多种配置方法，这里仅介绍其中最简单的一种。

下面这个例子将 Agent 机器上的 `/tmp` 目录挂载到了容器中的 `/data` 路径上，并在该目录中创建了一个名称为 `“hello”` 的文件，向其中写入了一些内容。

```
{
  "id": "marathon-volume-demo",
  "cmd": "echo 'Hello marathon!!' >/data/hello; sleep 3600",
  "cpus": 0.1,
  "mem": 8.0,
  "instances": 1,
  "container": {
    "type": "MESOS",
    "docker": {
      "image": "alpine"
    },
    "volumes": [
      {
        "containerPath": "/data",
        "hostPath": "/tmp/",
        "mode": "RW"
      }
    ]
  }
}
```

启动这个应用，在运行它的 Agent 上可以找到 `“/tmp/hello”` 文件，其中内容为 `“Hello marathon!!”`，说明指定的目录被挂载到容器中了。这个特性对 Mesos 的 Docker 容器化方式也同样适用，举一个比较实际的例子，以下这个 Json 文件可以用来快速创建一个 Docker

镜像仓库。

```
{
  "id": "docker-registry",
  "instances": 1,
  "cpus": 0.5,
  "mem": 1024.0,
  "disk": 128,
  "container": {
    "docker": {
      "type": "DOCKER",
      "image": "registry:latest",
      "network": "BRIDGE",
      "parameters": [],
      "portMappings": [
        {
          "containerPort": 5000,
          "hostPort": 0,
          "protocol": "tcp",
          "servicePort": 5000
        }
      ]
    }
  },
  "volumes": [
    {
      "hostPath": "/local/path/to/store/packages",
      "containerPath": "/storage",
      "mode": "RW"
    }
  ],
  "env": {
    "SETTINGS_FLAVOR": "local",
    "STORAGE_PATH": "/storage"
  },
  "ports": [0]
}
```

在这个应用描述文件中，容器中用于存储镜像数据的“/storage”目录被挂载到了容器外的 Agent 节点目录。这里用 `env` 属性给应用的运行上下文添加环境变量，它适用于各种类型的 Marathon 应用。

4.3.7 Marathon-LB 负载均衡

通过 Marathon 在 Mesos 集群部署的应用可能包含多个实例副本，它们运行的节点甚至端口都是在启动后才能确定的，如何让其他应用或是外部用户能够访问呢？在 Kubernetes 中提供了 Ingress 这样的对象，而在 Marathon 中也有相应解决方案：Marathon-LB。它由 Mesosphere 公司提供，并托管在单独的 GitHub 仓库中^①。

在 Marathon-LB 开始流行以前，还有一个叫作 Bamboo 的类似项目，它们的原理比较相似，都是基于 HAProxy 提供负载均衡能力。不过 Bamboo 不是 Mesosphere 公司的项目，随着 Marathon-LB 的成熟，它逐渐超过 Bamboo 成为主流的选择。

Marathon-LB 本质上是一些监听 Marathon 管理的应用，然后自动更新 HAProxy 配置的 Python 脚本，它的部署很简单。而相比之下，HAProxy 的部署和配置则复杂许多。为此，Mesosphere 公司维护了一个内置 HAProxy 的 Marathon-LB 免配置 Docker 镜像，并且推荐使用这种方式进行启动，如下所示。

```
$ docker run -d --name="marathon-lb" \
  --privileged --net=host -e PORTS=9090 \
  mesosphere/marathon-lb:v1.4.3 sse \
  --marathon http://172.31.31.164:8080 \
  --health-check \
  --group external
```

建议将 Marathon-LB 部署在 Mesos 集群之外的节点上，以方便进行流量和访问策略的管理，并为其预留足够的网络带宽和必要的端口。启动这个容器要用到 `--privileged` 和 `--net=host` 参数，因为该容器中的 HAProxy 需要操作节点的 IPTables 规则，并直接在节点的网卡上建立路由。PORTS 环境变量指定了 HAProxy 的 API 服务监听的端口。实际上，HAProxy 需要占用主机的 80、443、9090 和 9091 端口，其中的 80 和 443 分别作为对外的 HTTP 和 HTTPS 访问入口，9090 端口用于提供 API，而 9091 端口可以在没有设置域名和服务端口的情况下直接使用应用名称访问应用。

这个命令中还包含了 Marathon-LB 脚本启动所需的一些参数。其中，`sse` 表示使用监听 Marathon 的事件 API 来获得应用状态变化的信息，另一种方式是 `poll`，它通过定期查询的方式来获得应用的变化，后者只是为了兼容早期版本的 Marathon，通常不再使用了。`--marathon` 参数指定监听的 Marathon 服务地址，`--health-check` 参数表示只将流量负载均衡到健康检查通过的后端实例，`--group` 参数用于在 Mesos 集群中有多个 Marathon-LB

^① <https://github.com/mesosphere/marathon-lb>

实例时，区分应用要注册到哪个负载均衡器。通常推荐在一个集群中为内网和外网访问分别提供独立的负载均衡实例。

启动 Marathon-LB 的容器后，使用浏览器访问这个节点 9090 端口的“/haproxy?stats”地址，例如 <http://52.79.176.208:9090/haproxy?stats>，将看到 HAProxy 状态的页面，如图 4-10 所示。

HAProxy version 1.6.10, released 2016/11/20

Statistics Report for pid 700

> General process information

pid = 700 (process #1, nproc = 1)
uptime = 58.0801m12s
system limits: memmax = unlimited; ulimit-n = 100037
maxsock = 100037; maxconn = 60000; maxpipes = 0
current conn = 1; current pipe = 0; conn rate = 1/sec
Running tasks: 1/11; idle = 100 %

active UP
active UP, going down
active DOWN, going up
active or backup DOWN
active or backup DOWN
active or backup DOWN for maintenance (MAINT)
Note: "NOLEAF" or "DRAIN" = UP with load-balancing disabled.

Display option:

- Scope:
- Hide "DOWN" servers
- Refresh now
- CSV export

External resources:

- Prometheus site
- Upstream (v1.6)
- Online manual

Note: "COLBY:DRAIN" = UP with load-balancing disabled.

state		Queue				Session rate				Sessions				Bytes				Errors				Warnings				Server								
		Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LaTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Rate	Rate	Status	LastChk	Wght	Act	Bck	Chk	Down	Downs	Thrtle
Frontend					1	1	-	1	1	10 000																OPEN								
Backend		0	0	0	0	0	0	0	0	1 000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1m12s UP		0	0	0				0

marathon_http_in

Queue		Session rate				Sessions				Bytes				Errors				Warnings				Server												
		Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LaTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Rate	Rate	Status	LastChk	Wght	Act	Bck	Chk	Down	Downs	Thrtle
Frontend					0	0	-	0	0	0	0	0	10 000													OPEN								

marathon_http_appid_in

Queue		Session rate				Sessions				Bytes				Errors				Warnings				Server												
		Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LaTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Rate	Rate	Status	LastChk	Wght	Act	Bck	Chk	Down	Downs	Thrtle
Frontend					0	0	-	0	0	0	0	0	10 000													OPEN								

marathon_http_appid_in

Queue		Session rate				Sessions				Bytes				Errors				Warnings				Server												
		Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LaTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Rate	Rate	Status	LastChk	Wght	Act	Bck	Chk	Down	Downs	Thrtle
Frontend					0	0	-	0	0	0	0	0	10 000													OPEN								

图 4-10 HAProxy 的 Web 页面

Marathon-LB 将动态部署到 Agent 上未知端口的应用映射到固定节点的固定端口上，在负载均衡后端的多个实例副本会轮流响应用户请求。为了验证这个特性，一种简单可行的办法是使用一个能够获取并返回当前实际运行节点 Hostname 的服务，然后使用 Marathon 来部署它，比如下面这个例子。

```
$ cat << EOF > marathon-lb-demo.json
{
  "id": "marathon-lb-demo",
  "cpus": 0.5,
  "mem": 64.0,
  "instances": 2,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "flin/whoami",
      "network": "BRIDGE",
      "portMappings": [
        {
          "containerPort": 8000,
          "servicePort": 10080,
```



```

        "protocol": "tcp"
      }
    ]
  },
  "healthChecks": [{
    "protocol": "HTTP",
    "path": "/",
    "portIndex": 0,
    "timeoutSeconds": 10,
    "gracePeriodSeconds": 10,
    "intervalSeconds": 2,
    "maxConsecutiveFailures": 10
  }],
  "labels": {
    "HAPROXY_GROUP": "external",
    "HAPROXY_0_VHOST": "demo.caas.com"
  }
}
EOF

```

在这个应用描述中使用的“flin/whoami”镜像内置了一个监听 8000 端口，并向请求者返回自己运行容器主机名的小程序。**instances** 属性指定为这个应用创建两个实例副本。在描述文件的后半部分，**healthChecks** 是指定了一个监控后端服务是否正常运行的检查探针，通常来说，使用 Marathon-LB 负载均衡的服务都应该加上健康检查，以便 Marathon 能够自动去除不正常的副本，这样 Marathon-LB 就不会将请求路由到无法响应的节点上，关于应用的健康检查将在第 4.3.12 小节中专门介绍。

portMappings 属性指定了一个 **servicePort** 属性，这个属性告诉 Marathon-LB 在负载均衡器增加目标端口为 10080 路由规则。

描述文件的末尾部分为应用赋予了两个标签，其中“HAPROXY_GROUP”标签是实现 HAProxy 自动配置的关键。Marathon-LB 只有监听到存在“HAPROXY_GROUP”标签，且标签的值与自身启动时 **--group** 的值一致时，才会在 HAProxy 中自动创建或修改相应的路由规则。“HAPROXY_0_VHOST”标签指定了一个服务绑定的域名，其中的数字 0 表示它与第一个服务端口的访问入口相关联，它与 Kubernetes 的 Ingress 中配置的绑定域名作用类似，如果用户拥有这个域名，可以为域名添加一条指向 Marathon-LB 节点地址的 A 记录，然后就可以使用这个域名访问应用提供的服务了。

使用 **dcos** 工具创建这个例子，如下所示。

```
$ dcos marathon app add marathon-lb-demo.json
```


假设 Marathon-LB 服务部署的节点 IP 地址是 52.79.176.208, 现在有 3 种方法能够访问这个应用。

- 使用服务端口

直接访问 Marathon-LB 节点的 10080 端口, 如下所示。

```
curl http://52.79.176.208:10080
```

- 使用域名

如果能够绑定域名则将域名解析到 Marathon-LB 节点, 然后直接访问域名, 如下所示。

```
curl http://demo.caas.com
```

若无法修改域名解析, 可直接访问 80 端口, 并使用 Host 请求头指明要访问的域名, 如下所示。

```
curl -H "Host: demo.caas.com" http://52.79.176.208
```

- 使用 9091 端口

即使用户没有配置域名和服务端口, 依然可以通过 Marathon-LB 的 9091 端口访问部署的服务。此时需要添加 X-Marathon-App-Id 请求头, 值为要访问的应用名称, 如下所示。

```
curl -H "X-Marathon-App-Id: /marathon-lb-demo" http://52.79.176.208:9091
```

使用任意一种方式连续多次访问应用的负载均衡地址, 请求会被平均路由到两个后端实例上。输出内容打印的是容器内部的主机名, 如下所示。

```
$ curl http://52.79.176.208:10080
I'm d1ea520b787d
$ curl http://52.79.176.208:10080
I'm 86ea8c10b142
$ curl http://52.79.176.208:10080
I'm d1ea520b787d
$ curl http://52.79.176.208:10080
I'm 86ea8c10b142
```

需要指出的是, portMappings 是 Docker 容器任务特有的属性, 对于 Mesos 容器的任务, Marathon-LB 会使用 ports 或 portDefinitions 属性中的指定端口作为服务端口。不过由于缺少端口映射功能的支持, 为了让 Marathon-LB 能正确地路由请求, 需要将 Mesos 容器中服务监听的端口修改为与服务端口一致, 对于 “flin/whoami” 这个镜像, 可以定义 “WHOAMI_PORT” 环境变量来达到这个目的, 如下所示。

```
{
  "id": "marathon-lb-demo",
  "cpus": 0.5,
  "mem": 64,
  "instances": 2,
  "container": {
    "type": "MESOS",
    "docker": {
      "image": "flin/whoami",
    }
  },
  "healthChecks": [{
    ...
  }],
  "labels": {
    "HAPROXY_GROUP": "external",
    "HAPROXY_0_VHOST": "demo.caas.com"
  },
  "env": {
    "WHOAMI_PORT": "31080"
  },
  "ports": [ 31080 ],
  "requirePorts": true
}
```

注意在 `env` 属性中不能使用“`$PORT0`”这样的端口变量，需要直接写明实际的值。

Marathon-LB 同样支持使用 HTTPS 地址，对后端实例的选择性路由，以及为每个应用路由配置更复杂的负载均衡规则，可以参考其文档获得更多信息。

4.3.8 Mesos-DNS 域名服务

Mesos-DNS 是 Mesosphere 公司提供的另一个 Mesos 集群附加服务，代码同样开源在 GitHub 上^①。它提供域名解析和基于应用名称的服务发现功能，与 SwarmKit 或 Kubernetes 中自动依据服务名称为每个服务建立域名映射的能力十分相似。

作为域名解析服务，Mesos-DNS 的角色与互联网中 DNS 差不多，只不过它的域名解析规则不是人工配置的，而会根据集群中的 Marathon 或 Aurora 等框架部署的常住任务自动形成。Mesos-DNS 将这些在集群中运行的应用赋予域名，并通过 DNS 查询协议将域名

^① <https://github.com/mesosphere/mesos-dns>

解析成实际的服务地址。

从实现的角度上看, Mesos-DNS 不会直接与服务框架进行通信, 而是定期检索 Mesos Master 中所有运行的框架和正在运行的任务状态, 并为这些任务生成 DNS 记录, 包括 A 记录与 SRV 记录。与其他的通用分布式 DNS 服务相比, 例如 SkyDNS 或者 Consul, Mesos-DNS 的设计更加简单, 它自身没有状态、心跳检查、持久化储存和分布式一致性机制, 因为它是为 Mesos 定制的, 而在 Mesos 及其服务框架里已经实现了必要的容错性和服务生命周期的管理。

与其他 Golang 的服务一样, 部署 Mesos-DNS 只需下载它的二进制文件并运行, 如下所示。

```
$ curl -Lo mesos-dns https://github.com/mesosphere/mesos-dns/releases/download/v0.6.0/mesos-dns-v0.6.0-linux-amd64
$ chmod +x mesos-dns
$ sudo mv mesos-dns /usr/local/bin/
```

在启动 Mesos-DNS 服务前, 需要准备一个配置文件, 可以将它放到系统的“/etc”目录下, 如下所示。

```
$ cat << EOF | sudo tee /etc/mesos-dns.conf
{
  "zk": "zk://172.31.31.164:2181/mesos",
  "listener": "172.31.32.153",
  "port": 53,
  "domain": "mesos",
  "resolvers": ["114.114.114.114", "114.114.115.115"],
  "refreshSeconds": 60,
  "ttl": 60,
  "timeout": 5,
  "httpon": true,
  "dnson": true,
  "httpport": 8123,
  "externalon": true,
  "SOAMname": "ns1.mesos",
  "SOARname": "root.ns1.mesos",
  "SOARefresh": 60,
  "SOARetry": 600,
  "SOAExpire": 86400,
  "SOAMinttl": 60,
  "IPSources": ["netinfo", "mesos", "host"]
}
EOF
```

其中开头的几项配置相对比较重要。`zk` 指定 Mesos Master 的地址, 可以使用 ZooKeeper 记录的地址和路径表示。`listener` 和 `port` 指定 DNS 服务运行在哪个 IP 地址和端口, 通常使用标准的 DNS 端口: 53。`domain` 是生成的顶级域名, 可以是任意字符串, 它会影响生成的域名地址, 建议避免使用常见的顶级域名, 如 `com`、`org`、`cn` 等, 以免产生混淆。`resolvers` 是上级 DNS 域名服务器, 如果查询的域名不是 Mesos 服务域名 (比如互联网域名), Mesos-DNS 会通过这里指定的域名服务器去查询并返回结果。

同样建议使用 Systemd 管理 Mesos-DNS 服务, 如下所示。

```
$ cat << EOF | sudo tee /etc/systemd/system/mesos-dns.service
[Unit]
Description=Mesos DNS Service

[Service]
ExecStart=/usr/local/bin/mesos-dns \
    -config=/etc/mesos-dns.conf
Restart=always

[Install]
WantedBy=multi-user.target
EOF

$ systemctl enable mesos-dns
$ systemctl start mesos-dns
```

Mesos-DNS 只会自动生成必要的域名记录, 但它并没有什么特殊的魔法让所有节点自动使用它来解析域名。为了让 Mesos 集群中的节点能够通过服务域名相互访问, 必须在每个从节点上修改“`/etc/resolv.conf`”文件, 在首行增加相应的 `nameserver` 记录。假设 Mesos-DNS 部署的节点 IP 地址是 172.31.32.153, 此操作可以使用 `sed` 命令完成, 如下所示。

```
$ sudo sed -i '1s/^/nameserver 172.31.32.153\n /' /etc/resolv.conf
```

Mesos-DNS 注册的域名是对所有框架的, 地址规则为“<服务名>.<框架名>.<配置文件的 domain 值>”。比如前一小节在 Marathon 部署的 ID 为“`marathon-lb-demo`”服务域名是“`marathon-lb-demo.marathon.mesos`”。可以在配置过“`resolv.conf`”文件的主机上查询, 如下所示。

```
$ nslookup marathon-lb-demo.marathon.mesos
Server:      172.31.32.153
Address:     172.31.32.153#53
```



```
Non-authoritative answer:
Name:    marathon-lb-demo.marathon.mesos
Address: 172.31.38.204
```

由于配置了上级 DNS 服务器，在 Mesos-DNS 上同样可以解析集群外部的地址，如下所示。

```
$ nslookup baidu.com
Server:    172.31.32.153
Address:   172.31.32.153#53

Non-authoritative answer:
Name:      baidu.com
Address:   123.125.114.144
```

此外，由于 Docker 默认采用主机的“`resolv.conf`”文件内容创建容器中的“`resolv.conf`”域名服务器配置文件，因此 Mesos-DNS 生成的域名不仅对 Mesos 容器有效，对于使用 Docker 启动的服务也是有效的。可以在 Agent 节点上启动一个 Docker 容器验证，如下所示。

```
$ docker run --rm busybox \
    ping --count=1 marathon-lb-demo.marathon.mesos
PING marathon-lb-demo.marathon.mesos (172.31.38.204): 56 data bytes
64 bytes from 172.31.38.204: icmp_seq=0 ttl=64 time=0.190 ms
1 packets transmitted, 1 packets received, 0% packet loss
```

4.3.9 服务依赖和编组

在 SwarmKit 和 Kubernetes 中都有服务编排的概念。在 Marathon 中则提供了有些相似的编组（Group）功能来实现对应用的依赖和部署关联的管理。

Json 格式的应用描述文件天生具有树形的组织结构。Marathon 可以在描述文件中定义应用分组，将应用组作为这个树的枝干，而应用作为这个树的叶子。然后使用 `dependencies` 属性定义它们之间的依赖关系。

在下面这个例子中，ID 为“demo”的区块是一个应用组，而 ID 为“redis”和“webapp”的区块是两个应用，它们部署在 Marathon 中的完整 ID 实际上分别为“/demo/redis”和“/demo/webapp”，如下所示。

```
$ cat << EOF > marathon-group-demo.json
{
  "id": "demo",
```



```
"apps": [
  {
    "id": "redis",
    "cpus": 1,
    "mem": 1024,
    "disk": 1024,
    "instances": 1,
    "container": {
      "type": "MESOS",
      "docker": {
        "image": "my-private-repo.com/redis"
      }
    }
  },
  {
    "id": "webapp",
    "cpus": 1,
    "mem": 512,
    "instances": 3,
    "container": {
      "type": "MESOS",
      "docker": {
        "image": "my-private-repo.com/app"
      }
    },
    "dependencies": [
      "/demo/redis"
    ]
  }
]
}
EOF
```

注意在“webapp”这个应用中的 **dependencies** 属性，它指定了当前应用依赖于部署路径为“/demo/redis”的应用。这个应用组的结构如图 4-11 所示。

一旦设置了依赖关系，在创建这个应用组的时候，Marathon 会保证这个应用组中的应用按照依赖顺序进行创建。比如上例中会先创建“/demo/redis”应用，然后再创建“/demo/webapp”应用。同理，当对应用组进行扩容或者缩容时，如果组中的应用有依赖关系，那么先扩容被依赖的应用，再扩容依赖的应用，如果是缩容的话则相反。当删除应用组时，如果组中的应用有依赖关系，先删除被依赖应用的实例，然后删除依赖应用的实例。

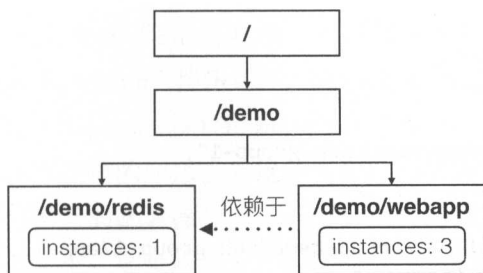


图 4-11 应用组结构

应用组的部署需要发送 POST 请求到 Marathon 的 `/v2/groups` 这个 API，若通过 `dcos` 工具执行，应该使用 `dcos marathon group` 命令组下面的子命令，例如创建这个应用组，如下所示。

```
$ dcos marathon group add marathon-group-demo.json
```

删除名称是“demo”的顶级应用组，如下所示。

```
$ dcos marathon group remove demo
```

如果部署时遇到“AppDefinition must either contain one of 'cmd' or 'args', and/or a 'container'.”这样的错误，说明在调用时误用了部署应用而不是应用组的 API。

当应用组部署完成后，它在 Marathon 的 Web 界面上会展示成一个文件夹的样式，并标明整个应用组使用的资源总量，如图 4-12 所示。

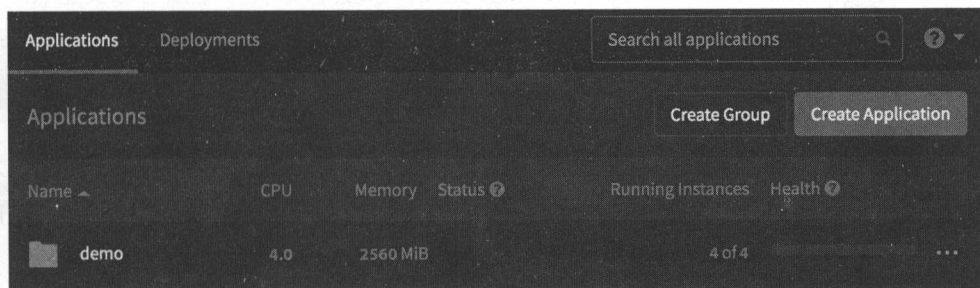


图 4-12 在 Marathon 中部署的应用组

应用组可以嵌套更多的应用组，从而形成复杂的应用部署结构。而且，依赖不但可以是应用与应用之间的，应用也可以依赖另一个应用组，这种情况下，只有被依赖的应用组中的应用全部创建之后，这个应用才可以被创建。下面这个简化的应用描述文件展示了这种结构，其中的应用组和应用 ID 使用了完整路径 ID 的表示形式，它与之前例子中的简写形式 ID 是等效的。

```
{
  "id": "/group-demo",
  "groups": [
    {
      "id": "/group-demo/sub-group-1",
      "apps": [
        {
          "id": "/group-demo/sub-group-1/app-1",
          "cmd": "..."
        },
        {
          "id": "/group-demo/sub-group-1/app-2",
          "cmd": "..."
        }
      ]
    },
    {
      "id": "/group-demo/sub-group-2",
      "apps": [
        {
          "id": "/group-demo/sub-group-2/app-3",
          "cmd": "...",
          "dependencies": [
            "/group-demo/sub-group-1"
          ]
        }
      ]
    }
  ]
}
```

需要指出的是，Marathon 对应用所定义的依赖关系只有在同一个应用描述文件中才有效。如果一个应用依赖了来自其他描述文件创建的应用，即使被依赖的应用没有被提前创建，当前应用仍然可以创建成功，它所定义的 **dependencies** 属性内容将被忽略。

4.3.10 应用升级

在 Marathon 中，应用升级指的是应用参数的变更。这包括更改执行的命令、更改 Docker 或 Unified Container 应用的镜像以及对其他运行参数的修改。通过 Marathon 的 Web 界面、dcos 的命令行工具、API 以及 SDK 都可以执行应用的升级。

在页面上对应用进行升级的操作很简单，进入具体应用的详细信息页面，单击名为

“Configure” 的 Tab 页，然后单击“Edit”按钮，对应用的配置信息进行更改后，单击蓝色的“Change and deploy configuration”按钮即可。Marathon 会自动创建一个部署操作，在后台进行服务的升级。部署（Deployment）是 Marathon 中对一次应用变更的抽象。部署操作具有独占性，一个正在进行变更的应用会拒绝所有后续的部署操作，此时如果有新的部署操作需要执行，要么等待当前的部署操作完成，要么明确地回滚正在进行的部署，这确保了每次部署都是原子性的变更，避免因用户“一边升级一边扩容”等并行变更导致无法预知的结果。

使用 `dcos` 命令行的 `dcos marathon app update` 命令可以对应用配置进行更改。例如将应用的副本数量增加到 4 个，如下所示。

```
$ dcos marathon app update marathon-cmd-demo instances=4
```

或者修改应用执行的命令，如下所示。

```
$ dcos marathon app update marathon-cmd-demo cmd="while [ true ]; do echo 'HTTP/1.0 200 OK\r\nContent-Length: 7\r\n\r\nHello Marathon' | nc -l -p $PORT0; done"
```

对于一些嵌套式的属性，更改时需要提供完整的新版本内容，不能只写需要更改的部分。例如更改应用使用的容器镜像版本，如下所示。

```
$ dcos marathon app update marathon-unified-demo \
  container='{ "type": "MESOS", "docker": { "image": "python:3.5.3-alpine" } }'
```

有些文章介绍了一种很方便的“全量更新应用”方式，即直接在 Json 描述文件中进行修改，然后将整个应用描述内容传递给 Marathon 执行更新，如下所示。

```
$ dcos marathon app update marathon-docker-demo < docker-demo.json
```

或是：

```
$ cat docker-demo.json | dcos marathon app update marathon-docker-demo
```

请注意，这种更新方式的实际效果与其给人的感觉是有差别的：它其实只会更新配置文件中修改或增加的属性，若配置文件里删除了某些属性，执行更新后，这些属性并不会从 Marathon 的应用里被删除，因此并非真正的“全量更新”。

造成这个问题的根本原因在于，`dcos marathon app update` 命令实际是调用了 Marathon 的“PUT /v2/apps”这个 API。而这个 API 本身的实现就与其语义不符，是对应用属性进行“局部更新”，本应使用 HTTP 的“PATCH”操作，却使用了“PUT”。而其 API 里对应用并没有提供一个真正的“PUT”全量更新操作。这可以认为是 Marathon 最初设计 API 时遗留的 BUG。总之，无论是使用 `dcos` 工具还是 Marathon API 进行应用更新时，都

应该避免由此引起的问题。

当应用的任何配置发生更改并提交后，Marathon 会自动重启应用的所有副本，使这些更改生效。不过，Marathon 默认的服务升级方式乍一看会有点奇怪，它先启动与原先应用相同个数的新版本副本，等到这些副本全部启动完成后，再删除原有的应用副本。这种做法配合 Marathon-LB 的自动路由切换，可以确保应用在升级过程中不会发生中断，即采用的是“不离线部署”，且新旧版本的整体切换是在最后的一瞬间完成的，两个版本几乎没有并存的时间（实际上由于新副本之间的启动完成时间有一定的差异，还是有短暂的新旧版本并存时间）。这种执行方式对于规模较小的服务是可行的，但对于有几十或上百副本的应用而言，升级的过程需要加倍使用集群资源，若资源不够，甚至会导致整个升级过程卡在中间，无法继续也无法撤销（这不是开玩笑，撤销部署时会产生一个回滚操作，相当于另一次部署过程，当整个集群资源不足时，撤销部署的操作也可能会发生卡壳），出现这种情况时可先强行将应用副本数减少（这会打断正在进行的部署），执行升级后再扩大副本数。

好在 Marathon 的服务升级过程其实是可以配置的，在应用描述中的 `upgradeStrategy` 属性可以对两个参数进行调整。

- `maximumOverCapacity` 表示升级过程中最大的超额副本数，用于告诉 Marathon 在应用升级时，为了避免服务中断，可以临时增加多少额外的副本。它会影响增加新副本的数目。
- `minimumHealthCapacity` 表示升级过程中保持的最小健康副本数，用于确保 Marathon 在应用升级时始终保留一部分对外提供服务的副本。它会影响删除旧副本的速度。

这两个参数的值都只能为 0~1 之间的小数（包括 0.0 和 1.0）。Marathon 在升级应用时，总是先根据 `maximumOverCapacity` 的值创建出足够数量的新版本应用副本，然后再根据 `minimumHealthCapacity` 的值逐步删除旧版本的应用副本。下面分别具体讨论。

1. `maximumOverCapacity`

在默认情况下，`maximumOverCapacity` 的值为 1。表示启动与原先应用相同数量的副本。例如应用有 8 个副本，则升级时先创建 8 个新版本的应用副本，再逐步删除旧版本的副本。使用这种方式升级应用的速度较快，但在重启过程中会占用较多的资源。

当 `maximumOverCapacity` 的值为 0 时，表示重启应用时不会先创建额外副本，而是一边删除旧的副本，然后创建相应数目的新副本，依次迭代，直到升级完成。用这种方式升级应用的速度相对较慢，但在重启过程中不会出现资源过多占用的情况。

当 `maximumOverCapacity` 的值大于 0、小于 1 的时候，表示先创建相应比例数量的新版本副本。例如应用有 8 个副本，`maximumOverCapacity` 的值为 0.4，则先创建 3 个新版本

的副本 ($0.4 \times 8 = 3.2$, 四舍五入), 然后开始逐步删除旧副本, 增加更多的新副本。这种方式在升级速度和资源占用之间进行了折中。

2. minimumHealthCapacity

在默认情况下, `minimumHealthCapacity` 的值为 1, 此时的升级过程是, 先根据 `maximumOverCapacity` 数量创建一定数量的新版本副本。只有成功创建一个新版本副本, 才能删除一个旧版本副本, 保持整体副本个数始终不少于设定的副本个数, 直到所有副本替换完成。使用这种方式升级应用的速度比较慢, 但由于在整个过程中, 应用副本数只会增加、不会减少, 因此对服务的稳定性影响最低。

当 `minimumHealthCapacity` 的值为 0 的时候, 升级时会将这个应用程序下的所有旧副本直接删掉, 然后等待新版本副本创建完成。用这种方式升级应用的速度很快, 但不推荐在实际生产环境中使用, 因为它会导致服务的短时间中断。

当 `minimumHealthCapacity` 的值大于 0、小于 1 的时候, 则升级过程介于前两者之间, Marathon 会确保一定比例的可用副本数量, 直到所有副本替换完成。这种方式是在升级速度和对服务的影响之间进行了折中。

值得指出的是, `minimumHealthCapacity` 参数的“可用副本数量”对于有“`healthChecks`”定义的应用而言指的是“启动完成且通过健康检查”的副本, 而对于没有“`healthChecks`”定义的应用而言, 则仅仅代表“启动完成”的副本。因此建议在可能的情况下, 总是为应用添加健康检查的属性。

现在将之前的 Unified Container 例子稍加修改, 使得它的升级过程更合理, 如下所示。

```
$ cat << EOF > marathon-unified.json
{
  "id": "marathon-unified-demo",
  "cmd": "python3 -m http.server ${PORT0}",
  "cpus": 0.5,
  "mem": 64.0,
  "instances": 4,
  "container": {
    "type": "MESOS",
    "docker": {
      "image": "python:3.5.1-alpine"
    }
  },
  "ports": [ 0 ],
  "healthChecks": [{
    "protocol": "HTTP",
```

```
        "path": "/",
        "portIndex": 0
    }],
    "upgradeStrategy": {
        "minimumHealthCapacity": 0.5,
        "maximumOverCapacity": 0.3
    }
}
EOF
```

应用在 Marathon 中的每次部署都会自动被保存为一个历史版本。使用 “/v2/apps/<应用 ID>/versions” 这个 API 可以查看到，通过 `dcos` 工具也可以方便地查找。应用的历史版本是使用部署时间标记的，越靠上的记录版本越新，如下所示。

```
$ dcos marathon app version list marathon-lb-demo
[
  "20XX-XX-XXT14:13:22.479Z",
  "20XX-XX-XXT12:10:31.994Z",
  "20XX-XX-XXT18:08:01.606Z"
]
```

当发现当前运行的应用版本功能有问题，或由于新版本应用没有通过健康检查而使得部署过程卡在中间无法进行时，可以使用 Marathon 的回滚功能，让应用恢复为以前的一个健康版本。

在 Marathon 中，回滚应用就是执行一次只有 `version` 属性的应用更新操作。从本质上来说，它就相当于使用指定版本的应用描述文件重新执行一次部署。不过它与普通的部署还有一点不同：当一个应用正在进行一个未完成的部署操作时，普通的部署请求都会被阻塞，唯独执行回滚操作的部署请求会中断当前的部署，并重新开始回滚版本的部署操作，如下所示。

```
$ dcos marathon app update marathon-lb-demo version="20XX-XX-XXT12:10:31.994Z"
```

此外，Marathon 还提供了蓝绿部署的支持。这是一种比较高级的、不中断服务的部署方法，可以进一步实现 A/B 测试和灰度发布等功能。蓝绿部署的具体步骤在 Marathon 的文档中有详细描述^①，其过程需要配合负载均衡器进行操作，Marathon-LB 提供了相应的 “`zdd.py`” 辅助脚本^②，可以比较方便地实施部署操作，这里就不再展开。

① <https://mesosphere.github.io/marathon/docs/blue-green-deploy.html>

② <https://github.com/mesosphere/marathon-lb/blob/master/zdd.py>

4.3.11 调度约束

虽然 Mesos 对集群的资源做了足够的抽象,但在部署应用时可能会有一些资源倾向性,比如希望一批磁盘 I/O 密集型的任务调度到有固态硬盘的机器上,此时就需要一种在调度上的选择机制来满足特定的场景,这种机制被称为调度约束 (Constraints)。比如使用节点的标签,把任务指定分配到具有一组有特定标签的节点上,或者是使用主机名等节点原本的属性进行资源标定和筛选。Mesos 本身是不做这类事情的,不过 Mesos 上的许多服务框架都有类似的功能,Marathon 也不例外。

在介绍 Mesos Agent 启动参数时,已经提到过 `--attributes` 这个参数,它用来在节点启动的时候,为节点添加一些特殊的键值标签。这些标签会作为 Mesos 为服务框架提供的资源 Offer 的内容之一传递给服务框架,这样 Marathon 之类的服务框架就有机会对节点进行选择了。Marathon 的应用调度约束同样是在应用描述里定义的,对应的属性是 `constraints`,格式如下所示。

```
"constraints": [{"<标签>", "<操作>", "<操作参数>"}]
```

约束值由三部分组成,其中第一部分为匹配的标签名称,除了 Agent 节点启动时自定义的标签,还有一个系统标签“hostname”表示 Agent 节点的主机名。第三部分的操作参数是可选的,根据第二部分具体操作类型决定是否需要。第二部分定义了约束的类型,它定义了 Marathon 对资源的选择方式,目前支持六种操作,其中有些操作的含义很难从其命名猜测到,列举如下。

1. UNIQUE

UNIQUE 操作告诉 Marathon 在指定标签值相同的每类节点上最多只能运行一个应用副本。比较常用的模式是将“hostname”用作标签,这使得该应用的副本在每个节点上最多不超过一个,如下所示。

```
"constraints": [{"hostname", "UNIQUE"}]
```

2. MAX_PER

MAX_PER 是 UNIQUE 操作的扩展,它告诉 Marathon 在指定标签值相同的每类节点上最多只能运行指定个数的应用副本。以下这个约束使得应用在每个区域最多部署两个副本。

```
"constraints": [{"zone_id", "MAX_PER", "2"}]
```

UNIQUE 相当于参数为 1 的 MAX_PER 操作。

3. GROUP_BY

GROUP_BY 与 UNIQUE 操作有点类似，它告诉 Marathon 将任务在通过指定标签值形成的分组之间，尽可能均匀地分散开。

假设集群被划分为多个区域，每个区域上的节点使用 zone_id 标识。在下面这个约束中，如果集群里 zone_id 的种类数比该应用的副本数更多，它相当于 UNIQUE 操作，每个区域最多只会会有一个副本。但如果该应用的副本数比集群里 zone_id 的种类数更多，则每个区域可能有超过一个副本，但依然会均匀地按区域分配，如下所示。

```
"constraints": [["zone_id", "GROUP_BY"]]
```

Marathon 是通过当前已经获取到的 Offer 来识别集群中有哪些 zone_id 的，但有时可用 Offer 的量并不全（例如当前某个区域的资源正好全部被用完，没有空闲的 Offer），此时 Marathon 可能会对区域数量误判，将副本重复部署到相同的区域上。为此可用使用一个参数告诉 Marathon，指定标签一共有几个可取的值，如下所示。

```
"constraints": [["zone_id", "GROUP_BY", "3"]]
```

GROUP_BY 有点像非强制性的 UNIQUE 操作，对于实施应用的高可用部署十分方便。

4. CLUSTER

CLUSTER 操作相当于标签匹配，它告诉 Marathon 将应用只部署在具有特定标签的 Agent 节点上。在应用有特殊的硬件要求，或者为了将副本部署在同一区域而降低通信延迟时很有用。下面这个约束会将应用副本仅部署在有固态硬盘的节点。

```
"constraints": [["disk_type", "CLUSTER", "SSD"]]
```

5. LIKE

LIKE 操作接受一个正则表达式作为参数，它告诉 Marathon 将应用只部署在具有特定标签且标签值符合指定表达式的 Agent 节点上，并尽可能均匀分布。下面这个约束会让副本仅部署在 1~3 号区域。

```
"constraints": [["zone_id", "LIKE", "zone-[1-3]"]]
```

6. UNLIKE

UNLIKE 操作和 LIKE 相反，它告诉 Marathon 将应用只部署在具有特定标签但标签值不符合指定表达式的 Agent 节点上，并尽可能均匀分布。下面这个约束会让副本不要部署在 7~9 号区域。

```
"constraints": [["zone_id", "UNLIKE", "zone-[7-9]"]]
```

4.3.12 健康检查

前文已经讲述了一些应用健康检查配置的例子，当应用的某些副本无法通过健康检查时，Marathon 会将其重启以尝试使它恢复正常。在服务升级时，配置了健康检查的服务可以更准确地反映新版本的副本是否已经正常运行，从而为 Marathon 控制升级节奏提供支持。此外，应用副本的健康检查结果还能影响 Marathon-LB 的路由规则，使得用户的请求不会被路由到不健康的副本上，从而确保发生局部故障时，对外提供的服务依然正常。

Marathon 使用 `healthChecks` 属性定义应用的健康检查方式，这个属性的值是一个列表，因此可以为每个应用定义多个检查点。对于每个检查点，Marathon 支持三种不同的健康检查方式。

1. HTTP

具体可以分成“HTTP”“HTTPS”“MESOS_HTTP”和“MESOS_HTTPS”四个子类。它们都使用基于 HTTP 访问的方式验证服务的可用性。返回的 HTTP 状态码若在 200~399 之间，则认为副本运行正常。四种方式的不同之处在于使用 HTTP 协议或者 HTTPS 协议，以及从什么地方访问服务的副本。对于前两种方式，访问的请求是从 Marathon 的运行节点发出的，而对后两种方式，访问请求是从运行副本的 Mesos Agent 节点发出的。

具体的应用描述举例如下所示。

```
"healthChecks": {
  "protocol": "HTTP",
  "portIndex": 0,
  "path": "/api/health",
  "gracePeriodSeconds": 300,
  "intervalSeconds": 60,
  "timeoutSeconds": 20,
  "maxConsecutiveFailures": 3
}
```

2. TCP

检查指定 TCP 端口是否有监听。具体可以分成“TCP”和“MESOS_TCP”两个子类。两者的差异同样在于访问副本的是 Marathon 节点还是 Mesos Agent 节点。应用描述举例如下所示。


```
"healthChecks": {  
  "protocol": "MESOS_TCP",  
  "portIndex": 0,  
  "gracePeriodSeconds": 300,  
  "intervalSeconds": 60,  
  "timeoutSeconds": 20,  
  "maxConsecutiveFailures": 0  
}
```

3. COMMAND

运行特定命令，如果命令正常结束则认为副本健康。在这种方式下，访问请求总是在运行副本的 Mesos Agent 节点执行的。需要注意的是，如果要执行的命令里包含双引号，需要用三个反斜杠转义。在读取文件时，前两个反斜杠经过转义成为一个有效的反斜杠符，而第三个反斜杠用于转义双引号，在命令执行时，转移过一次的反斜杠和双引号组合会再次被转义，最后得到一个有效的双引号。应用描述举例如下所示。

```
"healthChecks": {  
  "protocol": "COMMAND",  
  "command": {"value": "/bin/bash -c \\""/bin/check.sh $PORT0\\""},  
  "gracePeriodSeconds": 300,  
  "intervalSeconds": 60  
}
```

其中的具体参数对于不同的检查方式大同小异，以下列举常用的参数和适用的检查类型。

- **protocol**: 指定具体使用的检查方式。
- **portIndex**: 对于 HTTP 和 TCP 检查方式，指定检查应用声明的第几个端口。
- **path**: 对于 HTTP 检查方式，指定访问的路径。
- **command**: 对于 COMMAND 检查方式，指定执行的具体命令内容。
- **gracePeriodSeconds**: 可选参数，默认值为 300。指定在应用启动多久以后监控检查开始生效，对于一些启动速度较慢的应用十分有用，单位是 s。
- **intervalSeconds**: 可选参数，默认值为 60。指定两次检查之间的时间间隔，单位是 s。
- **timeoutSeconds**: 可选参数，默认值为 20。指定连接或命令的超时时间，单位是 s。
- **maxConsecutiveFailures**: 可选参数，默认值为 3。指定连续多少次检查失败后才认为副本不可用，这是为了避免因偶然的网络抖动等原因导致检查结果不准确。

4.4 使用 Chronos

4.4.1 部署 Chronos

Chronos 框架最初由 Airbnb 公司开发，而后开源并共享给 Mesos 社区，现在代码托管在 GitHub 的 Mesos 组织下^①。目前 Chronos 已经是 Mesos 集群定时任务框架的现实标准，在许多基于 Mesos 架构的数据中心集群中具有与 Marathon 相当的地位。Mesosphere 当前正在重新设计一个新的定时任务框架：Metronome。不过在它真正成熟起来之前，Chronos 依然是久经考验的首选方案。

作为集群的定时任务管理框架，Chronos 的功能与 Linux 系统的 Crontab 十分相似。在分布式集群中，定时任务最终需要运行在特定的某个节点上，而每个节点的资源都是有限的，为此 Chronos 基于 Mesos 提供了更多适应于集群环境的能力，例如依据资源选择节点、失败任务自动重试、任务的资源配额限制、指定任务应运行的最长时间以及允许任务之间相互依赖等。和 Marathon 一样，Chronos 也支持使用 Mesos 容器或 Docker 容器运行任务，并提供基于 Json 格式的 Restful API，用于创建、修改、删除和手动触发任务。

与 Marathon 一样，Chronos 可以部署多个节点形成高可用集群。为了在多个节点之间共享集群的任务状态信息，Chronos 使用 ZooKeeper 来存储任务的数据。

由于 Chronos 主要使用 Scala 语言编写并依赖 Mesos 的“libmesos.so”文件，部署之前需要先安装 Mesos 和 1.8 以上版本的 JDK，使用以下命令即可启动。

```
$ export MESOS_NATIVE_LIBRARY=/usr/local/lib/libmesos.so
$ java -jar chronos.jar --master zk://172.31.31.164:2181/mesos \
    --zk_hosts 172.31.31.164:2181 \
    --zk_path "/chronos"
```

不过，Chronos 在 GitHub 发布的版本只提供源代码包，而没有预编译好的 Jar 文件，因此倘若想运行它的 jar 包，只能自己从源码编译，比较麻烦。好在 Chronos 的开发者在 Docker Hub 维护了 Chronos 的 Docker 镜像，官方也建议通过 Docker 镜像的方式启动这个框架，如下所示。

```
$ docker run -d --name=chronos \
    -e PORT0=8080 -e PORT1=8081 \
    -p 8080:8080 -p 8081:8081 \
```

^① <https://github.com/mesos/chronos>

```
mesosphere/chronos:v3.0.1 \  
--master= zk://172.31.31.164:2181/mesos \  
--zk_hosts=172.31.31.164:2181 \  
--zk_path "/chronos"
```

其中 PORT0 环境变量用于指定 Chronos 的 Web 界面和 Restful API 的端口，而 PORT1 环境变量用于指定 LibProcess 协议库的 Protocol Buffer 接口的端口。服务启动的三个参数分别表示 Mesos Master 的地址（可以使用 Mesos 在 ZooKeeper 注册的路径表示）以及存储 Chronos 任务数据的 ZooKeeper 集群地址和路径。

启动后在运行节点的 8080 端口可以看到 Chronos 的界面，如图 4-13 所示。

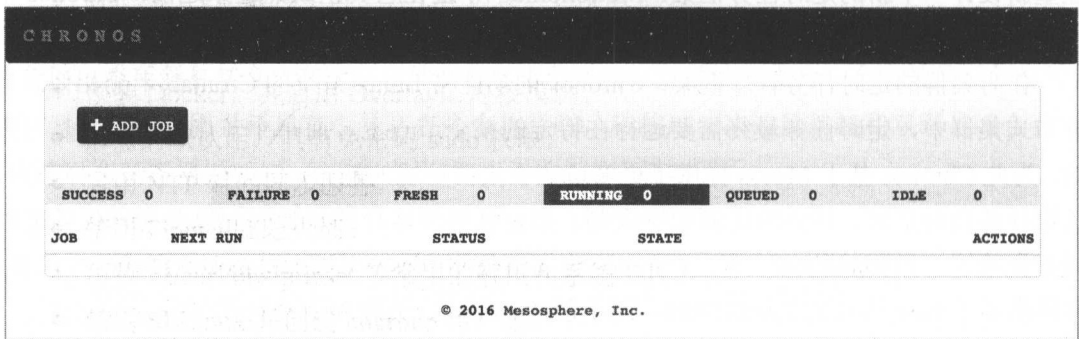


图 4-13 Chronos 的 Web 界面

4.4.2 定时表达式

在 Chronos 中定义计划任务时，需要使用一种三段式的表示方式。例如“R10/2017-04-16T19:20:30.45+01:00/P1Y2M3DT4H5M6S”，其中，斜线作为分隔的位置，具体格式如下所示。

<重复次数>/<开始时间>/<重复频率>

因此例子中的表达式分解出来就是以下这三个部分。

- 重复次数：R10。
- 开始时间：2017-04-16T19:20:30.45+01:00。
- 重复频率：P1Y2M3DT4H5M6S。

下面依次解释这三个部分的含义。

1. 重复次数

表示该事件需要重复执行多少次，用字母 R 开头，然后接一个数字表示重复次数。例如“R10”表示重复 10 次。没有数字，只有一个字母“R”则表示无限重复。

2. 开始时间

使用 ISO-8601 格式时间戳，也就是“YYYY-MM-DDThh:mm:ss.sTZD”这种表示法。

其中 YYYY 是年份，MM 是月份，DD 是日期，用固定的字母 T 表示日期与时间的分隔位置，hh 是小时，mm 是分钟，ss.s 则是秒数，可以有小数点精确到 ms（但实际上毫无意义，Chronos 执行时间本来就没有这么高的精度），最后的 TZD 表示时区，例如东 8 区用“+08:00”表示。

如果这部分留空则表示任务在提交后立即开始计时。这种用法常见于一些重复频率较高的场景，例如每隔几分钟就要执行一次的任务，精确写明开始时刻的意义不大。

3. 重复频率

格式为“P<年月日>T<时分秒>”，其中开头的 P 和中间的 T 是固定的，而从左至右的 Y、M、D、H、M、S 依次表示年、月、日、时、分、秒，它们都是可以省略的。

前面示例中的“P1Y2M3DT4H5M6S”实际上表示“每隔 1 年 2 个月零 3 天又 4 小时 5 分钟 6 秒”的时候重复执行一次，这是个很长的时间。不过，实际上 Chronos 会忽略最后“秒”的部分，因为它的时间间隔精度只能支持到分钟，也就是说，最小的任务执行间隔就是分钟。

对于没有日期的情况，依然需要保留格式中的 P 和 T 字母，例如“PT1M”表示每隔一分钟执行一次任务。

4.4.3 创建定时任务

与 Marathon 一样，Chronos 使用 Json 格式的描述文件定义，其中包含一些必要的信息，包括任务名称、需要的资源量、定时周期、依赖关系和任务执行内容等。

最简单的一种 Chronos 任务是不依赖于其他任务的独立任务，类似于每周固定时间发送提醒邮件或每天晚上进行数据库备份这样的任务。用户可以通过界面或 API 的方式来创建它。

通过 Web 界面来创建定时任务非常直观，打开 Chronos 的 Web 界面，单击“Add Job”按钮，在下拉框中选择“scheduled”，在弹出的对话框中填入任务的信息，如图 4-14 所示。

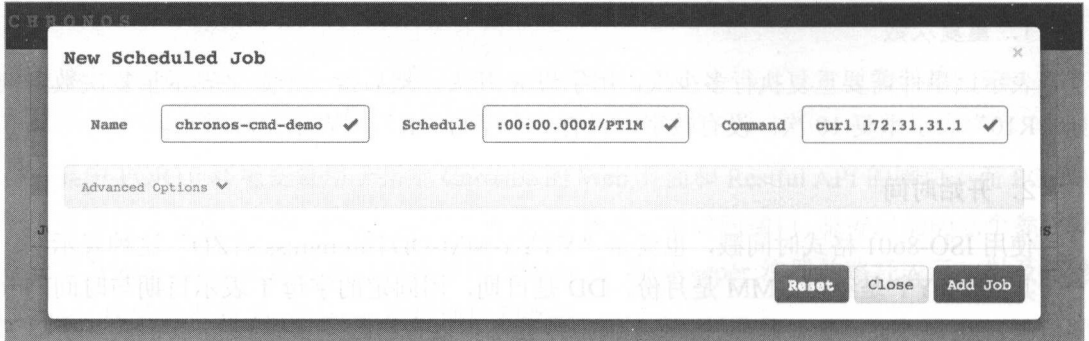


图 4-14 通过 Chronos 界面创建定时任务

这个任务会每隔一分钟向指定节点的 6000 端口发送一个 HTTP 包。

在实际场景中，推荐大家学习使用 API 的方式创建任务。一方面，这样可以更灵活地与其他运行工具进行集成。另一方面，Chronos 的 API 提供了比 Web 界面更丰富的功能，例如使用 Unified Container 特性，目前这在 Web 界面上是无法配置的。

创建 Chronos 任务的 API 需要用 HTTP POST 方式调用，可以使用以下别名来简化命令。

```
$ alias curl-post='curl -X POST -H "Content-type: application/json"'
```

首先创建一个任务描述文件，如下所示。

```
$ cat << EOF > chronos-cmd-demo.json
{
  "schedule": "R//PT1M",
  "name": "chronos-cmd-demo",
  "owner": "linfan.china@gmail.com",
  "description": "check an url every 2mins",
  "command": "curl 172.31.164:6000",
  "owner": "LinFan",
  "cpus": "0.1",
  "mem": "1",
  "disk": "0"
}
EOF
```

然后用 POST 方式访问与 Web 界面相同的 8082 端口，路径是 “/v1/scheduler/iso8601”。使用下面的命令将读取 “chronos-cmd-demo.json” 文件中的内容，并放到请求里发送给 Chronos。

```
$ curl-post -sL http://52.79.176.188:8082/v1/scheduler/iso8601 \
-d @chronos-cmd-demo.json
```


然后在这个任务访问的目标节点上运行一个简单 HTTP 服务，如下所示。

```
$ docker run --rm --tty --net=host python:3-alpine python3 -m http.server 6000
```

可以观察到这个服务每隔 2 分钟会收到一个访问请求，证实 Chronos 中部署的定时任务已经正常运行了。这里必须加上 `--tty` 参数，否则容器中输出的内容不会立即回显到控制台上。

在 Chronos 的任务描述中还有一个比较常用的属性是 `environmentVariables`，它用于在任务运行的环境中增加环境变量，如下所示。

```
"environmentVariables": [
  { "name": "END_POINT", "value": "172.31.31.164:6000" }
]
```

直接访问 `/v1/scheduler/jobs` 路径可以获得当前以及部署的所有定时任务详情，它的输出是未格式化过的 Json，可以使用 `jq` 命令将其处理成更加可读的输出显示，如下所示。

```
$ curl -sLX GET 52.79.176.188:8082/v1/scheduler/jobs | jq '.'
```

使用 DELETE 请求访问 `/v1/scheduler/job/<任务名称>` 路径将删除指定的定时任务，如下所示。

```
$ curl -sLX DELETE 52.79.176.188:8082/v1/scheduler/job/chronos-cmd-demo
```

那么，当 Chronos 调度任务的时候，它会如何选择节点呢？现在通过下面这个例子来观察一下。

```
$ cat << EOF > chronos-docker-demo.json
{
  "schedule": "R/PT1M",
  "name": "chronos-docker-demo",
  "owner": "linfan.china@gmail.com",
  "description": "write task date into node which it run",
  "retries": 2,,
  "command": "date >> /var/log/datatime.log"
  "container": {
    "type": "DOCKER",
    "image": "busybox:latest",
    "network": "BRIDGE",
    "volumes": [
      {
        "containerPath": "/var/log/",
        "hostPath": "/tmp",
        "mode": "RW"
      }
    ]
  }
}
```

```
}  
]  
,  
"cpus": "0.1",  
"mem": "1",  
"disk": "10"  
}  
EOF
```

当前 Chronos 已经能够支持使用 Docker 容器或 Mesos 容器运行容器镜像，上面的例子有意使用了镜像来创建任务以展示这种方式的用法。示例中的任务每分钟都会将当前时间写入到运行节点的临时目录内一个名为“datetime.log”的文件中。如果集群中有许多 Agent 节点，在运行一段时间后会发现在几乎每一个 Agent 节点的临时目录中都出现了“datetime.log”文件，并且写入的时间分布序列都是没有规律的。这证实了 Chronos 实际在每次执行任务时，是在集群中所有满足条件里随机选择一个作为运行节点的。

如果确实需要限制任务执行的节点，可以在任务描述文件中增加 **constraints** 属性，它的语法与 Marathon 的调度约束是一样的，如下所示。

```
"constraints": [["zone_id", "EQUALS", "zone-1"]],
```

4.4.4 定时任务的依赖

除了直接指定定时任务的执行周期，Chronos 还可以通过依赖关系建立灵活的定时任务流，如图 4-15 所示。其中 task1 任务配置了固定的执行周期，task2 与 task3 均依赖于 task1，它们会在每次 task1 执行完成后并行地执行，而 task4 同时依赖于 task2 与 task3，只有这两个任务都完成，它才会执行。相比 Marathon 的树形依赖（有向无环图），Chronos 的依赖关系是一个更复杂的有向图。

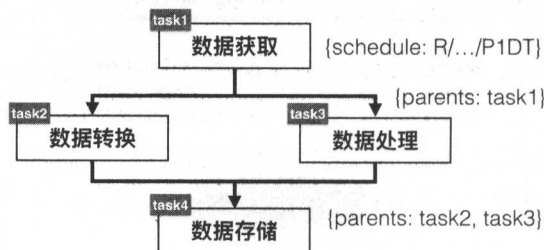


图 4-15 通过依赖关系管理任务

定义有依赖的任务与前一小节中介绍的方法并没有多少差别。需要注意的是，这类任

务没有指定执行周期的 `schedule` 属性，取而代之的是指定依赖关系的 `parents` 属性，它的值是一个依赖任务名称的列表，如下所示。

```
{
  "name": "task4",
  "description": "a job dependences on others",
  "parents": [
    "task2",
    "task3"
  ],
  "command": "...",
  ...
}
```

4.5 更多的 Mesos 服务框架

4.5.1 Mesos 服务框架的本质

在前面的内容里，已经介绍了 Mesos 的两层调度架构，并详细讲解了两个通用的服务框架：Marathon 和 Chronos。这可能会给大家造成一个误区，认为 Mesos 的服务框架就是调度其他服务的服务。

为了解 Mesos 中“服务框架”概念的初衷，现在再次回到 Mesos 诞生之初，Benjamin 博士设计这个东西的时候，想要解决的问题就是：怎样让应用程序更方便地运行在由许多分布式核心组成的计算机整列里。于是他设计了一个能将具体计算任务自动分发到不同的计算机单元上执行，然后把计算结果汇总回来的程序。接下来，为了让这个程序变得更通用，Benjamin 把进行运算逻辑的部分从复杂的底层调度代码分离出来，最终系统中底层调度的部分就成为了 Mesos 的雏形，这样就可以基于公共底层更快地编写出各种具备分布式运算功能的程序了。这些运行在 Mesos 之上，解决某些单一且具体的业务逻辑或科学计算问题的程序，就是最早的“服务框架”。这也是 Benjamin “数据中心操作系统”设想的原型：Mesos 架设在分布式硬件之上，成为一种专用内核，在内核之上可以像编写普通应用程序那样编写各种分布式应用程序。

后来 Benjamin 设计 Spark 时则是将编写分布式计算程序中的一些常用套路又归纳了一下，形成一个公用的基础计算程序。这开启了使“服务框架”来进行二次调度和简化开发

的思路，并形成调度器 (Scheduler) 和执行器 (Executor) 的划分。之后 Twitter 设计的 Aurora 框架则将这个思路进一步扩展到了更通用的领域，并随着 Spark 和 Hadoop 等能够运行于 Mesos 之上的通用计算框架逐渐流行，这种两级调度的理念逐渐固化了下来。后来的 Marathon 和 Chronos 等都已经属于“现代服务框架”之列了。但就本质来说，所谓的“服务框架”其实指的就是“能够直接运行在 Mesos 内核之上的应用程序”，它们通过 Mesos 设计的 API 接口来直接获取计算资源，并下发运行任务。

在 Mesos 源代码的“example”目录里还有很多简单的服务框架例子，包括使用蒙特卡洛方法计算圆周率以及只能运行一个 alpine 容器的服务框架等。这些框架都只能完成特定的一种任务，其中的一些可以在启动时接受用户输入的参数信息，从而对计算过程进行调整。这些程序或许从功能上并不应该被称为“框架”，因为并不具有多少可复用性，但实质上与 Marathon 等高级框架一样，都是运行在 Mesos 内核上的一种程序。

为了进一步看清 Mesos 框架的结构和本质，不妨简单了解一下 Mesos 的框架 API。

Mesos 的原始 API 主要分成三类，分别是调度器接口、执行器接口以及消息处理接口。设计一个服务框架，首先要从框架的调度器开始，调度器在启动时通过调度器接口将自己注册到 Mesos 中，然后像一个普通的服务程序运行。当服务框架有任务要运行时，不论它是被设计成只能执行固定的一种任务，还是像 Marathon 那样可以从外部配置灵活地执行各种任务，框架调度器都会通过消息处理接口向 Mesos 索取一定量的计算资源。Mesos 将各个节点上的可用计算资源以特定的数据格式（被称为 Offer）通过另一个消息处理接口发送给服务框架，由服务框架选择要或者不要，一旦服务框架选择接受该 Offer，则 Mesos 会被分配到的资源详细记录下来。然后调度器就可以把要执行的任务细节通过用于执行任务的调度器接口发给 Mesos，由 Mesos 放到已经分配给该任务的节点上去运行。

到目前为止，以上描述的这些步骤都是服务框架的调度器在与 Mesos 打交道，压根没有执行器什么事情。实际上，在服务框架中，并非必须要有执行器。前面提到的计算圆周率、随便运行一个容器等这些操作都可以由调度器直接下发命令完成。只不过在这种运行方式下，调度器把任务发给 Mesos 后，就任由 Mesos 发挥了，当任务运行结束，Mesos 会自动地回收任务的资源，然后通知调度器任务的结果（执行返回成功或失败）。在执行的中途，调度器无法与它的执行任务通信或发送额外的指令。因此对于更复杂一些的场景，比如在任务执行过程中，调度器希望能够干预运行状态或是希望任务能主动、及时地反馈执行的进展情况，就得让执行器出场了。服务框架的调度器在启动任务时除了直接给 Mesos 发命令，还可以提供一个 URL 地址，Mesos 收到这样的请求就会自动把地址指定的文件下载到目标 Agent 节点并运行，这个文件就是所谓的执行器。也就是说，执行器其实可以是

与调度器在代码层面无关的另一个单独程序，它实现了 Mesos 的执行器接口，因此能在执行期间通过 Mesos 接口找到启动它的调度器，并与之通信。另外，服务框架的组成部分不仅限于调度器和执行器，还可以有自己的 Web 界面，甚至是命令行工具和其他辅助组件，毕竟它实际上就是个与 Mesos 进行 API 通信的、独立运行的程序。

4.5.2 编写自己的 Mesos 服务框架

大致理解 Mesos 的框架原理以后，就可以自己动手写一个简单的 Mesos 服务框架。

原始的 Mesos API 是基于 Protocol Buffer 协议的，因此开发 Mesos 框架通常需要使用它的 SDK。Mesos 社区为 C++、Java、Python、Go 等主流语言均提供了专门的代码库^①。这里以 Java SDK 为例，简单介绍一下开发 Mesos 服务框架的大致过程。

首先需要在项目中引入 Java SDK 的依赖，如果使用 Maven 构建，可使用如下配置。

```
<dependency>
  <groupId>org.apache.mesos</groupId>
  <artifactId>mesos</artifactId>
  <version>1.1.1</version>
</dependency>
```

其他构建工具可参考这个依赖包的仓库说明页^②。为了方便程序运行，建议在打包时将所有依赖全部放进 Jar 包，这可以通过 maven-assembly-plugin 插件来实现。要是想实现一个自带 Web 界面的服务框架，不妨直接使用 Spring Boot 构建项目。它们都能使得生成的 Jar 包可使用“java -jar <包文件>”来执行。

然后创建一个自定义的类，实现 org.apache.mesos.Scheduler 接口，并补全该接口中的 10 个抽象回调方法的定义，如下所示。这里只讲解其中与执行任务相关的 resourceOffers 方法，其余的 9 个方法可暂时全部留空，具体作用可参看 SDK 文档。

```
package com.caas.demo.scheduler;

import org.apache.mesos.Scheduler;
import org.apache.mesos.Protos.ExecutorInfo;
import org.apache.mesos.Protos.CommandInfo;

public class DemoScheduler implements Scheduler {
```

① <http://mesos.apache.org/documentation/latest/api-client-libraries/>

② <https://mvnrepository.com/artifact/org.apache.mesos/mesos/1.1.1>


```
private int idCounter = 0;

String executorURL = "http://172.31.10.10/executor.jar";
String command = "java -jar executor.jar";

@Override
public void resourceOffers(SchedulerDriver schedulerDriver,
                           List<Protos.Offer> offers) {

    for (Protos.Offer offer : offers) {

        Protos.TaskID taskId = Protos.TaskID.newBuilder()
            .setValue(Integer.toString(idCounter++))
            .build();

        CommandInfo.URI executorUri =
            CommandInfo.URI.newBuilder()
                .setValue(executorURL)
                .setExtract(false)
                .build();

        CommandInfo commandInfo =
            Protos.CommandInfo.newBuilder()
                .addUris(executorUri)
                .setValue(command)
                .build();

        ExecutorInfo executorInfo =
            ExecutorInfo.newBuilder()
                .setExecutorId(Protos.ExecutorID.newBuilder()
                    .setValue("Executor " + taskId))
                .setCommand(commandInfo)
                .setName("Demo Executor")
                .setSource("java")
                .build();

        Executor executor =
            Protos.ExecutorInfo.newBuilder(executorInfo);

        Protos.TaskInfo task = Protos.TaskInfo.newBuilder()
            .setName("Task " + taskId)
            .setTaskId(taskId)
            .setSlaveId(offer.getSlaveId())
```

```

        .addResources(buildResource("cpus", 1))
        .addResources(buildResource("mem", 128))
        .setExecutor(executor)
        .build();

    launchTask(schedulerDriver, offer, task);
}
}

// 此处省略其余 9 个方法实现，其方法体内容均留空即可
}

```

每当集群中有可用的 Agent 节点资源时，Mesos 就会轮流调用各个服务框架的 `resourceOffers` 回调方法，将资源的 Offer 信息提供给框架。正常来说，框架应该根据当前是否有任务需要执行以及提供的 Offer 资源量是否满足需要来决定是否要启动一个任务。作为演示，这里省略了相应的判断操作，可以想象在这个服务框架启动后，可能会耗尽集群中所有可用的资源来不停地执行任务。目前暂且不考虑这样做可能带来的后果。这个方法首先遍历了每个 Offer 对象，然后创建了一个 `ExecutorInfo` 对象，其中包含了要执行的命令以及一个 URL。现在假设框架执行器的 Jar 包被存放在“`http://172.31.10.10/executor.jar`”这个内网服务器地址。当方法最终调用 `launchTask()` 操作时，Mesos 就会去下载指定的执行器文件，并在目标 Agent 上运行。

接着来实现程序的入口对象，在对象的主方法中创建了 `MesosSchedulerDriver` 对象，并调用它的 `run()` 方法，此时程序会进入服务框架的事件循环。直到框架收到异常信号或用户终止时，才会继续执行之后的 `stop()` 方法，然后结束整个程序，如下所示。

```

package com.caas.demo.scheduler;

import org.apache.mesos.MesosSchedulerDriver;
import org.apache.mesos.Protos;
import org.apache.mesos.Protos.FrameworkInfo;

public class Application {

    public static void main(String[] args) {
        String mesosAddr = args[0];
        FrameworkInfo.Builder builder = FrameworkInfo.newBuilder();
        builder.setFailoverTimeout(120000);
        builder.setUser("");
        builder.setName("Demo Framework");
        FrameworkInfo framework = builder.build();
    }
}

```

```
DemoScheduler scheduler = new DemoScheduler();

MesosSchedulerDriver driver = new
    MesosSchedulerDriver(scheduler, framework, mesosAddr);

int status = driver.run() == Status.DRIVER_STOPPED ? 0 : 1;
driver.stop();
System.exit(status);
    }
}
```

回到项目的根目录，使用 `mvn` 命令构建这个包，如下所示。

```
$ mvn clean package
```

这样一个简单的服务框架调度器就完成了，将构建出的包重命名为“`scheduler.jar`”，如下所示。

```
$ mv target/demo-scheduler-*.jar target/scheduler.jar
```

此时还没有准备好框架的执行器，先不必着急启动这个服务。

新建一个 Maven 项目，如下所示，同样在“`pom.xml`”文件中添加 Mesos SDK 依赖和 `maven-assembly-plugin` 插件。然后在项目中创建一个实现 `org.apache.mesos.Executor` 接口的类型，该接口包含 8 个抽象方法，这里只关注其中的 `launchTask` 方法，其余方法的实现内容均可留空。

```
package com.caas.demo.executor;

import org.apache.mesos.Executor;
import org.apache.mesos.ExecutorDriver;
import org.apache.mesos.Protos;

public class DemoExecutor implements Executor {

    @Override
    public void launchTask(ExecutorDriver executorDriver,
        Protos.TaskInfo taskInfo) {

        int replyCount = 10;
        while(replyCount > 0) {
            String reply = String.valueOf(replyCount--);
            executorDriver.sendFrameworkMessage(reply.getBytes());
            Thread.sleep(10000);
        }
    }
}
```

```

        Protos.TaskStatus status = Protos.TaskStatus.newBuilder()
            .setTaskId(taskInfo.getTaskId())
            .setState(Protos.TaskState.TASK_FINISHED)
            .build();
        executorDriver.sendStatusUpdate(status);
    }

    // 此处省略其余 7 个方法实现，其方法体内容均留空即可
}

```

当执行器启动后，Mesos 就会调用它的 `launchTask` 方法实现。在这个例子中，让执行器每隔 10s 给框架调度器回送一个数字，连续 10 次，然后退出。

在执行器程序入口对象的主方法中，使用 `MesosExecutorDriver` 对象将执行器注册到 Mesos 并且启动，如下所示。

```

package com.caas.demo.executor;

import org.apache.mesos.MesosExecutorDriver;

public class Application {
    public static void main(String[] args) throws Exception {
        MesosExecutorDriver driver = new MesosExecutorDriver(
            new DemoExecutor());
        int status = driver.run() == Status.DRIVER_STOPPED ? 0 : 1;
        System.exit(status);
    }
}

```

在项目的根目录，同样使用 `mvn` 命令进行构建，并将生成的包重命名为“`executor.jar`”，如下所示。

```

$ mvn clean package
$ mv target/demo-executor-*.jar target/executor.jar

```

将这个包放到内网能够获取的地方，例如 IP 地址为 172.31.10.10 的一个 HTTP 文件服务器。

最后，回到调度器程序的代码目录，使用以下命令启动它。

```

$ java -jar target/scheduler.jar zk://172.31.31.164:2181/mesos

```

调度器程序的参数是 Mesos 的 Master 服务地址，可以使用 ZooKeeper 存储的地址。

Mesos 于 1.0 版本发布以后，在原先 Protocol Buffer 的基础上提供了 HTTP 的通用 Restful API，使得用户现在可以不受 SDK 语言的限制，使用任意语言来快速开发服务框架。社区

中已经出现了一些基于新 API 的快速开发框架,对此感兴趣的读者可以关注 GitHub 上的这个 JavaScript 封装项目^①。

4.5.3 其他常见服务框架

虽然,理论上任何程序都可以被改造成 Mesos 的框架来运行,但这毕竟需要额外的适配改造工作。在实际的应用场景中,更常见的还是“业务程序——服务调度——资源调度”的两层调度+业务功能模式。也就是说,直接运行在 Mesos 上的服务框架往往是一些匹配许多通用场景的辅助程序,用来让普通程序更方便地利用 Mesos 提供的分布式运行功能。

除了本章介绍的 Marathon 和 Chronos,网络上还有许多现成服务框架,它们都是由企业或社区维护的。下面列举一些相对比较活跃的例子。

1. Spark

Spark 与 Mesos 一同出自加州大学伯克利分校的 AMP 实验室,是最早运行在 Mesos 上的通用的并行计算框架。事实上,Mesos 和 Yarn 一直是分布式计算领域里最广泛流行的资源调度平台,而 SwarmKit 和 Kubernetes 等在这个领域的市场都只能望其项背。

作为一种典型的分布式计算框架,Spark 支持批量技术、流式计算、数据预测、海量数据查询等许多应用场景。它将所有计算数据抽象成为 Dataset (包括经典的弹性分布式 Dataset 和新的结构化 Dataset)对象,并动态地将 Dataset 分发到各个节点上进行计算。Spark 可以在没有 Mesos 的情况下采用独占的方式管理计算资源,但基于 Mesos 运行 Spark 时,能够有效地与其他服务框架复用资源,从而使硬件资源利用率最大化。

由于数据对计算资源的使用通常都是高密度但间歇性的,在计算进行时占用大量 CPU、内存和存储,在计算结束后将资源又全部闲置,因此不仅是 Spark,主流的分布式计算框架几乎都提供了基于 Mesos 的服务框架实现,以便能够共享集群中的硬件,如 Hadoop、Storm、Flink 等。此处将它们作为同一类型的应用示例,不再逐一列举。

2. Kafka

Kafka 是由 LinkedIn 开源的一个分布式消息队列服务,经常被作为多个计算框架之间进行数据缓冲和格式解耦的中间件。Kafka 队列基于消息的发布和订阅通知,能在廉价的商用机器上做到单机每秒十万条以上消息的传输,并通过水平扩展增加队列带宽,即使对

^① <https://github.com/tobilg/mesos-framework>

TB 级以上的数据也能保证常数时间复杂度的访问性能。

Broker 是 Kafka 集群中消息服务节点的称呼。Kafka 的扩展特性使得它很适合在 Mesos 提供的分布式环境中使用，在通信流量增加时，通过 Mesos 的资源池，Kafka 可快速地动态增加 Broker 数量，当流量减少以后，关闭多余的 Broker，把资源空出来给其他框架使用。并且这些扩缩操作只需调用 Kafka 的 Mesos 调度器服务 API，完全免去手动部署和管理的麻烦。

相比其他的例子，Kafka 是一种典型的单一用途框架，不过由于它在数据计算领域的广泛使用，Mesos 社区专门维护了它的服务框架并保持着良好的支持。

3. Aurora

Aurora^①是 Twitter 公司开发并开源的通用任务运行框架，它同时支持常驻后台任务和定时计划任务，也已经支持 Mesos 和 Docker 容器的运行，功能上相当于 Marathon 和 Chronos 的集合。

Aurora 框架诞生于 2010 年，是最早的一批 Mesos 框架，早于 Marathon 和 Chronos 等后起之秀。不仅如此，Aurora 的定位是大型业务组织的任务调度系统，因此对系统的实用性和安全性的考虑都比较周全，支持命名空间，允许多个环境共存，此外还提供任务优先级抢占功能，从而支持在同一个集群中混合部署关键业务服务、测试任务和实验性任务。时至今日，Aurora 依然支撑着 Twitter 公司的许多关键业务。

即使从现在的角度来看，Aurora 在成熟度和功能性上也一点都不逊色于 Marathon 或 Chronos，但在流行度上却远不及后两者，其主要原因在于 Aurora 复杂的部署和过于陡峭的学习曲线。Aurora 曾一度只提供 Apache Thrift 框架的调用接口，而不是通用的 Restful API。而学习 Aurora 就像是在学一门新的编程语言，它提供了可复用的模板、模块化的设计，甚至支持 import 语法来引用各种系统组件，这种 DSL 式的配置语法明显提高了框架的入门槛。

4. Swan

Swan^②是国内的数人云公司开源的通用常驻任务调度框架。作为数人云数据中心解决方案的一部分，它的作用与 Marathon 相似，包括应用程序水平扩缩、滚动升级、版本回滚、健康检查、自动故障转移等核心功能。

① <https://github.com/apache/aurora>

② <https://github.com/Dataman-Cloud/swan>

相比于 Marathon, Swan 进行了许多有意思的功能改进。比如 Swan 内置了 Raft 选举模块, 从而不必依赖 ZooKeeper 等服务就可以实现高可用的部署, 这个思路与 SwarmKit 有异曲同工之妙。然后, Swan 还内置了服务发现和 DNS 模块, 相比依赖外置的 Marathon-LB 和 Mesos-DNS 实现的服务发现, 这种整体化方案部署更加简单和稳定。此外, Swan 增强了滚动更新功能, 允许应用部分实例更新, 从而实现类似 Kubernetes 通过修改 Label 将多个版本实例混合部署的灰度发布功能。

目前 Swan 服务框架还在开发中, 但十分值得期待。

5. Metronome

Metronome^①是由 Mesosphere 公司开发的定时任务框架, 并且已经内置在了 DC/OS 数据中心解决方案中, 目的是替代发展缓慢的 Chronos。

从功能上来看, Metronome 除了支持定时任务, 还支持手动触发的单次任务。它的最大特点其实是在代码层面复用了 Marathon 的许多模块, 从而使得二者在发展步调和功能更新上可以相互促进。它的使用和配置十分简单, 基于与 Marathon 相似的 Json 文件配置任务, 有兴趣的读者不妨尝试一下。

除此以外, 还有一些在 Mesos 的发展过程中值得一提的项目, 比如由社区发起的 Kubernetes on Mesos 和 Swarm on Mesos 项目, 以及 IBM 牵头的 kube-mesos-framework 项目, 它们试图在 Mesos 之上整合其他容器调度框架的功能, 从而真正地实现平台大统一。这些项目虽然由于种种原因最终没能获得广泛的关注, 却足以体现 Mesos 的巨大潜力。

4.6 DC/OS

4.6.1 DC/OS 简介

Mesosphere DC/OS 是一种新型的横跨数据中心或者云机器的操作系统, 它以开源的 Mesos 作为分布式内核, 整合了后台任务管理服务 Marathon、定时任务管理服务 Metronome、分布式文件系统 HDFS、应用打包和部署仓库 Universe 等核心基础设施, 以及一个功能齐全的命令行工具和 Web 操作界面。它就像是分布式系统中的 Ubuntu 那样的发行版, 已经

^① <https://github.com/dcos/metronome>

围绕内核增加了所有的系统服务和工具，是用户能够开箱即用的快速部署应用和容器。

DC/OS 数据中心操作系统发布于 2015 年 4 月，包含社区和商业两个版本，在最初的时候，社区版本仅支持部署在 AWS、GCE 或 Azure 上，并仅限于测试用途。整整一年后，Mesosphere 公司在 2016 年 4 月开源了社区版的 DC/OS，并提供了能够让用户在自建的数据中心环境部署 DC/OS 的工具，这使得 DC/OS 成为了 Mesosphere 公司能够正面对抗 Kubernetes 等开源集群平台的有力武器。

开源的社区版本提供了与企业版几乎相同的功能，两者的差异主要是在安全和认证模块上。例如社区版本没有提供 LDAP 或 Active Directory 等企业级的用户登录支持，取而代之的是基于 OAuth 协议的 GitHub、Google 和 Microsoft 账号登录。在用户权限方面，社区版也仅仅提供了所有用户都是管理员这种简单粗暴的方式，显然这些功能对于企业场景而言是无法满足的，但并不影响了解 DC/OS 的功能使用。本章的内容主要基于社区版 DC/OS 进行讲解。

图 4-16 展示了 DC/OS 的操作界面。

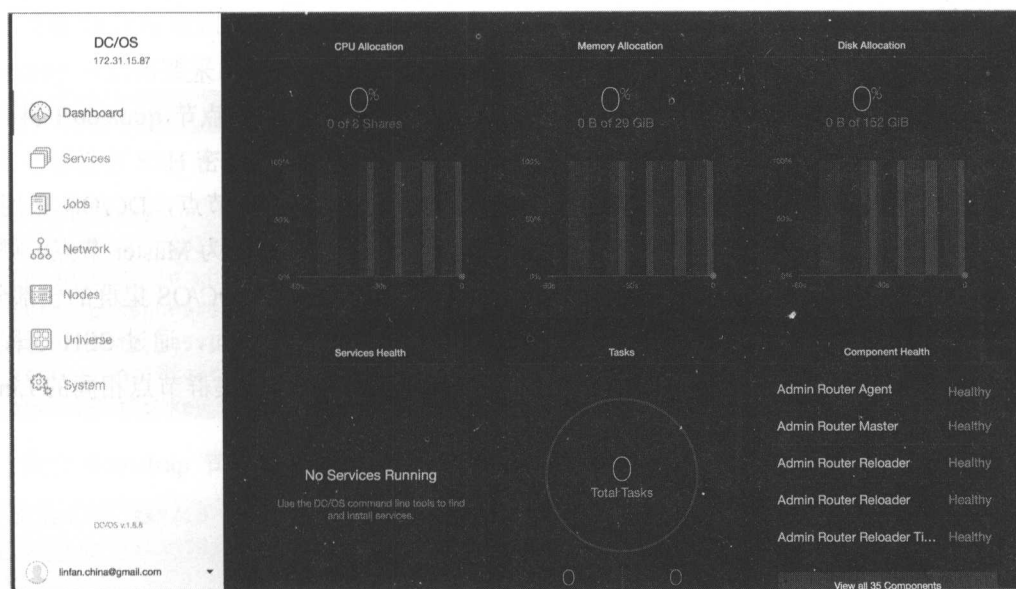


图 4-16 DC/OS 的操作界面

4.6.2 部署 DC/OS

DC/OS 提供了三种主要的安装途径，包括在本地环境使用 Vagrant 快速安装，直接在

AWS、GCE 或 Azure 中使用现成的模版来安装，以及下载 DC/OS 的通用安装包在任意集群中安装。

第一种方式将 DC/OS 的所有组件安装在一个本地虚拟机上，并且提供已经制作好的镜像，只需直接启动。这种方式安装的 DC/OS 不具有扩展性，仅仅适合演示。第二种方式基于特定的云主机平台，使用平台特有的初始化模板进行自动化安装。第三种方法是一种半自动的安装方式，可适用于任意的集群环境，下面主要介绍这种安装方式。

DC/OS 对于机器的 CPU、内存和系统组件有一些必须的要求，具体如下所示。

- 只支持 RHEL/CentOS 7.2 或 CoreOS 系统。
- Master 节点：4 核 CPU，32GB 内存，120GB 磁盘。
- Agent 节点：2 核 CPU，16GB 内存，60GB 磁盘。
- 预装 Docker，并启用 Overlayfs 内核模块。
- 节点的默认用户具有免密码 sudo 权限。
- 启用 NTP 时间同步服务。
- 禁用 Firewalld 防火墙。
- 预装 xz/curl/unzip/ipset 等常用的解压和系统工具。
- 禁用 SELinux 并创建 nogroup 用户组。
- 设置 LC_ALL 和 LANG 环境变量值为 en_US.utf-8。

本书采用 RHEL 7.2 系统进行讲解。首先要准备待安装的主机节点，DC/OS 采用和 Mesos 同样的 Master/Agent 主从结构，因此需要预先确定哪些节点作为 Master 节点、哪些作为 Agent 节点。除此之外，还需要一个 Bootstrap 节点，它不属于 DC/OS 集群的一部分，只在初始化集群的时候使用。Bootstrap 节点用于运行安装脚本或界面，通过 SSH 远程部署集群，对操作系统没有特殊要求，为了便于命令统一，可以采用与集群节点相同的 Linux 发行版。

在各个节点分别执行下面这些初始化命令。

关闭防火墙和 SELINUX，如下所示。

```
$ sudo systemctl stop firewalld
$ sudo systemctl disable firewalld
$ sudo sed -i s/SELINUX=enforcing/SELINUX=permissive/g \
/etc/selinux/config
```

安装系统工具，创建 nogroup 用户组，如下所示。

```
$ sudo yum install -y tar xz unzip curl ipset
$ sudo groupadd nogroup
```

开启内核 Overlay 支持，然后安装 Docker，如下所示。

```
$ sudo tee /etc/modules-load.d/overlay.conf <<- 'EOF'
overlay
EOF
$ sudo reboot

$ sudo tee /etc/yum.repos.d/docker.repo <<- 'EOF'
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/7/
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
EOF
$ sudo yum install -y docker-engine-1.11.2
$ sudo systemctl start docker
$ sudo systemctl enable docker
```

当前 DC/OS 建议使用 Docker 1.11 版本，这个版本相对比较稳定。由于在安装过程中，Bootstrap 节点需要采用 SSH 登录到集群的各个节点进行部署操作，在正式开始安装前，还需要将 Bootstrap 节点的 SSH 公钥添加到集群中每个节点的信任列表中。如果在 Bootstrap 节点上还没有 SSH 密钥对，可以用 `ssh-keygen` 命令生成一个，如下所示。

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/boot/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/boot/.ssh/id_rsa.
Your public key has been saved in /home/boot/.ssh/id_rsa.pub.
```

查看 Bootstrap 节点的公钥内容，如下所示。

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCMOfS...
```

把公钥内容添加到每个集群节点中用于执行安装脚本的用户的“`~/.ssh/authorized_keys`”文件末尾，用于执行安装脚本的用户应该是 root 或是具有免密码 `sudo` 的权限。

回到 Bootstrap 节点，下载 DC/OS 的安装工具包，如下所示。

```
$ curl -O https://downloads.dcos.io/dcos/stable/dcos_generate_config.sh
```

这个工具包提供了界面和命令行两种方式部署 DC/OS 集群（其中命令行方式还可细分

成简易模式和高级模式两种)，它们在本质上也没什么不同，都会经过预检查、部署依赖包、部署 DC/OS 和初始化配置几个步骤，这里介绍使用界面部署的方式。启动参数 `--web` 表示界面方式运行，`-v` 表示在部署过程中输出更多的详细信息，如下所示。

```
$ sudo bash dcos_generate_config.sh --web -v
```

然后通过浏览器访问 Bootstrap 节点的 9000 端口，就可以看到 DC/OS 的安装界面了。在界面上依次添加 Master 节点的 IP 列表、内部 Agent 节点的 IP 列表和对外 Agent 节点的 IP 列表。

其中 Master 节点个数通常是单数，建议使用 3 个或 5 个，以确保集群的高可用，最少的个数是 1 个，这种情况下的 Master 节点存在单点故障。内部 Agent 节点与对外 Agent 节点的区别主要在于节点上部署的服务是否对外暴露公网 IP 地址，这两类 Agent 在初始化后会划到不同的 Role 中，内部 Agent 属于集群的默认 Role（即用星号“*”表示的 Role），对外 Agent 节点则划到名为“slave_public”的单独 Role 中作为预留资源。因此在默认情况下，集群中的任务都是部署到内部 Agent 上的，这符合企业安全的原则，即绝大部分的服务是不需要对外访问的。而有些服务，例如 Marathon-LB，确实需要公网的 IP 地址，则可在部署描述中通过指定 `HAPROXY_GROUP`、`HAPROXY_0_VHOST` 等标签将其指派到特定的对外 Agent 上，通常在 DC/OS 集群中需要准备大量的内部 Agent 节点和少量的对外 Agent 节点。另外需要指出的是，Agent 节点所属的 Role 并非一成不变，DC/OS 提供了一个辅助脚本 `dcos_install.sh`，能够比较方便地转换 Agent 节点的角色。

其他的几个选项，Master Public IP 只需指定一个 Master 节点的 IP 地址，在有多个 Master 节点的情况下，可以指定任意的一个，这个 IP 会被作为 DC/OS 的默认 Web 界面入口地址使用。SSH Username 和 SSH Listening Port 需要指定的是集群节点的 SSH 用户名和 sshd 服务监听端口，通过 Web 界面安装 DC/OS 时要求所有待部署的集群节点都使用相同的 SSH 用户和端口。而在“Private SSH Key”一栏中需要填入 Bootstrap 节点的 SSH 私钥，这里不用担心密钥泄露的风险，因为 Bootstrap 节点仅仅在集群部署的过程中存在，当集群部署完成后可以关闭该节点，并将相应的 SSH 公钥从所有节点的信任列表中移除。也正是出于这样的考虑，通常不建议使用集群里的节点作为 Bootstrap 节点。

图 4-17 展示了一个只有单 Master 节点、单内部 Agent 和单对外 Agent 的最小集群配置。

在环境配置的部分，“Upstream DNS Servers”选项用于配置 DC/OS 内置的 Mesos-DNS 服务的上游域名服务器地址，DC/OS 默认使用了 Google 的域名服务器，可以根据实际情况更改。“IP Detect Script”选项需要提供一个能够检查节点内网 IP 地址的脚本，DC/OS 内置了几种云平台的获取 IP 脚本，并允许用户上传自定义的脚本文件。对于 RHEL/CentOS 7.2

系统，官方提供了一个供用户参考的自定义脚本内容，如下所示。

```
#!/usr/bin/env bash
set -o nounset -o errexit
export PATH=/usr/sbin:/usr/bin:$PATH
echo $(ip addr show eth0 | grep -Eo \
'[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}' | head -1)
```

The screenshot shows the 'Setup' page of the DC/OS configuration tool. It includes sections for Master and Agent IP lists, SSH configuration, and DC/OS Environment Settings.

Master Private IP List (Upload csv): 172.31.31.164

Agent Private IP List (Upload csv): 172.31.27.243

Agent Public IP List (Upload csv): 52.79.184.67

Master Public IP: 52.78.217.183

SSH Username: root

SSH Listening Port: 22

Private SSH Key (Upload):

```
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEA...
-----END RSA PRIVATE KEY-----
```

DC/OS Environment Settings

Upstream DNS Servers: 8.8.8.8, 8.8.4.4

IP Detect Script: Select an IP detect script

☒ Send Anonymous Telemetry ☒ Enable Authentication

图 4-17 DC/OS 的最小集群配置

这个内容的关键在于最后一行命令，可以在任意的节点上测试一下该命令，应当能显示出当前节点的 IP 地址。将以上内容保存到一个本地的文本文件中，然后选择下拉框中的“Custom Script”，上传脚本即可。配置妥当后，单击“Run Rre-Flight”按钮进入部署的预检查步骤，如图 4-18 所示。



图 4-18 DC/OS 的预检查步骤

这个过程相对较长，根据实际网络情况大概需要几十分钟到几个小时不等，所幸所有节点都是并行操作的，操作时间并不会因为节点数目增加而成倍增长。完成后，若一切正常，就会看到如图 4-19 所示的成功界面。

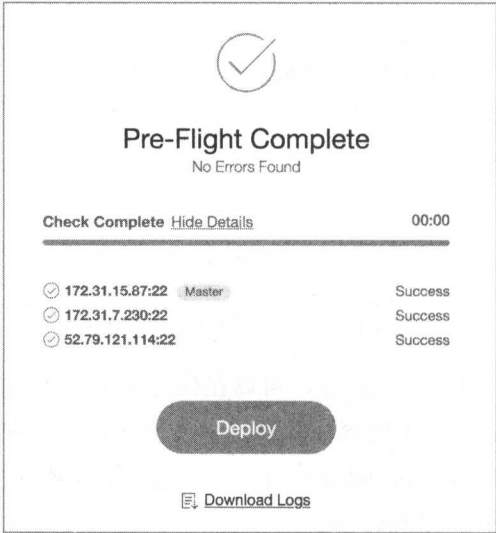


图 4-19 DC/OS 预检查成功

若检查出现问题，则会显示一个红色的叹号，单击“Show Details”链接将显示具体的错误原因，如图 4-20 和图 4-21 所示。

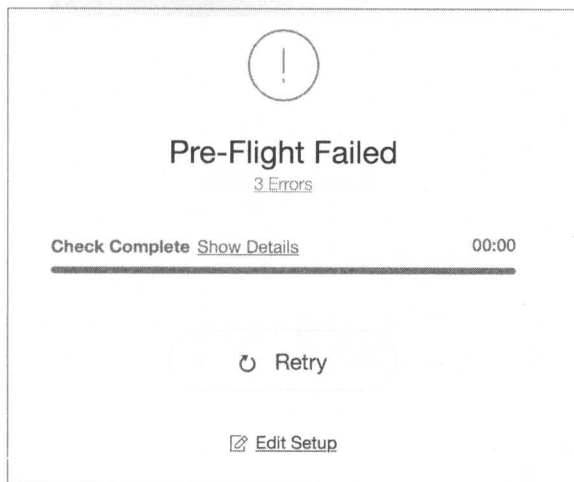


图 4-20 DC/OS 预检查失败

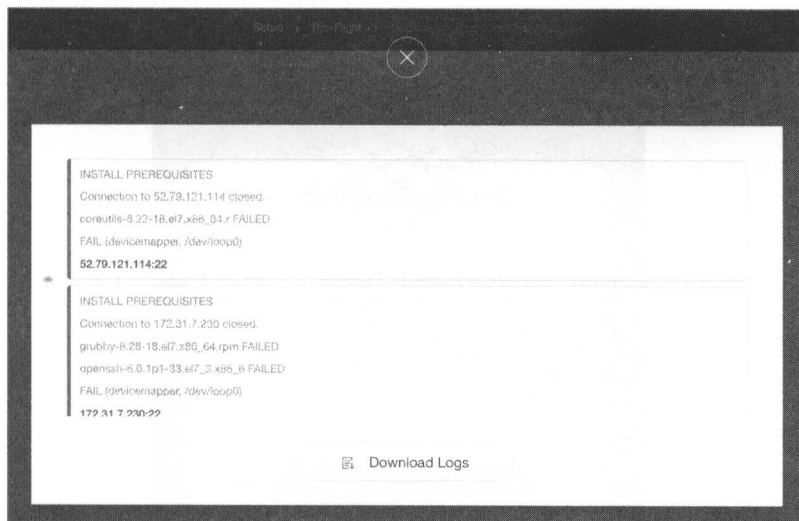


图 4-21 DC/OS 预检查错误详情

图 4-21 所示的是一个经常遇到的问题，原因是集群节点安装了 Docker，但没有启用 Overlayfs 模块。用户可以根据提示修复问题，然后单击“Retry”按钮重新检查。检查通过后，就可以单击“Deploy”按钮进行真正的部署了。

部署的过程主要是拷贝文件，通常几分钟就可以完成，出现错误的情况也比较少，具

体情况如图 4-22 和图 4-23 所示。



图 4-22 DC/OS 执行部署

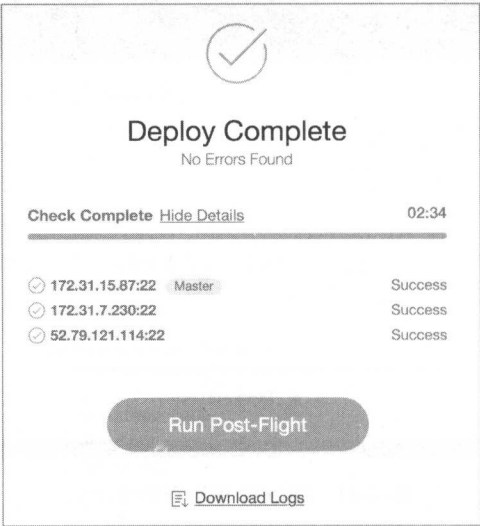


图 4-23 DC/OS 部署步骤完成

单击“Run Post-Flight”按钮，开始最后的检查工作。这个步骤通常只需要几十秒到几分钟时间，然后就可以看到最终部署成功的界面了，如图 4-24 所示。

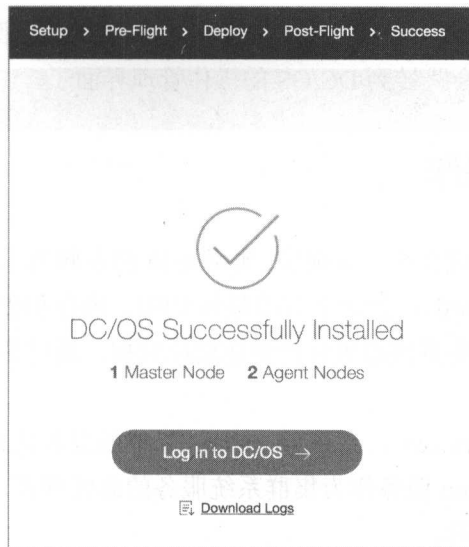


图 4-24 DC/OS 集群创建完成

此时单击“Log in to DC/OS”按钮会跳转到 DC/OS 的登录页面，社区版 DC/OS 仅支持使用 GitHub、Google 和 Microsoft 账号登录，如图 4-25 所示。

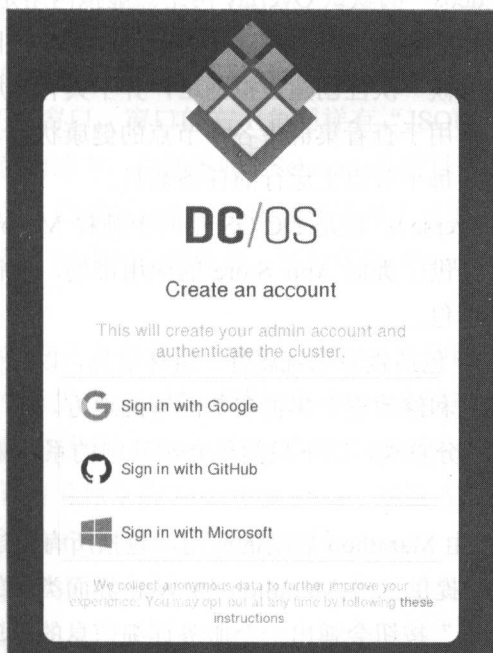


图 4-25 DC/OS 登录

以 GitHub 账号为例，单击页面上的“Sign in with GitHub”图标，会跳转到 GitHub 的账号登录页面，登录后就会跳转到 DC/OS 的操作管理界面了。

4.6.3 DC/OS 的操作

DC/OS 的操作界面分成 7 个主要板块，如图 4-16 的左侧所示，由上往下依次如下所示。

- 集群总览 (Dashboard)：包含集群的整体 CPU、内存和磁盘资源使用情况，当前运行任务数量，以及集群核心组件的健康监控数据。通过这个页面可以快速地了解集群当前的状态。
- 系统服务管理 (Services)：系统 Marathon 管理的 service 状态。在 DC/OS 中，提供了一个内置的 Marathon 服务作为集群系统服务的总管理者，其他所有的用户服务都应当由此服务进行管理。
- 系统任务管理 (Jobs)：系统 Metronome 管理的任务状态。包含定时任务和由事件触发的一次性任务，短生命周期的任务管理也是 DC/OS 的核心功能之一。
- 网络管理 (Network)：查看 DC/OS 的虚拟网络，通过 Web 界面安装的集群通常只会用一个名称为“dcos”的默认 Overlay 网络。虚拟网络是 DC/OS 中通过划定子网隔离，从而限制应用相互访问的措施，类似 SwarmKit 中的网络管理功能，但 DC/OS 当前只能在安装的时候一次性创建虚拟网络，并不具有实用性。
- 节点管理 (Nodes)：用于查看集群中各个节点的健康状态、CPU / 内存 / 磁盘资源使用情况以及当前在每个节点上运行的任务数目。
- 应用市场管理 (Universe)：这是 DC/OS 不同于纯粹 Mesos 加各种组件拼凑系统的一项重要功能，它提供了类似 App Store 的应用市场，使得用户能够很方便地找到集群支持的各种应用包。
- 系统配置 (System)：包括查看系统属性、组件信息，以及管理用户、管理应用仓库地址等功能，是查看和修改整个集群参数配置的地方。

这些面板的内容大多十分直观，以下只简单介绍其中的系统服务管理 (Services) 和系统任务管理 (Jobs) 面板。

Services 面板用于管理由 Marathon 启动的应用，包括所有通过 Universe 应用市场部署的应用都会出现在这里。它提供了与 Marathon 的 Web 界面类似的功能，在布局上也如出一辙。单击“Deploy Service”按钮会弹出一个服务详细信息的窗口，在窗口右上角同样有一个“JSON mode”切换器，如图 4-26 所示。在界面上填入服务的配置信息，单击“Deploy”，

就在集群中创建了一个新的常驻服务。

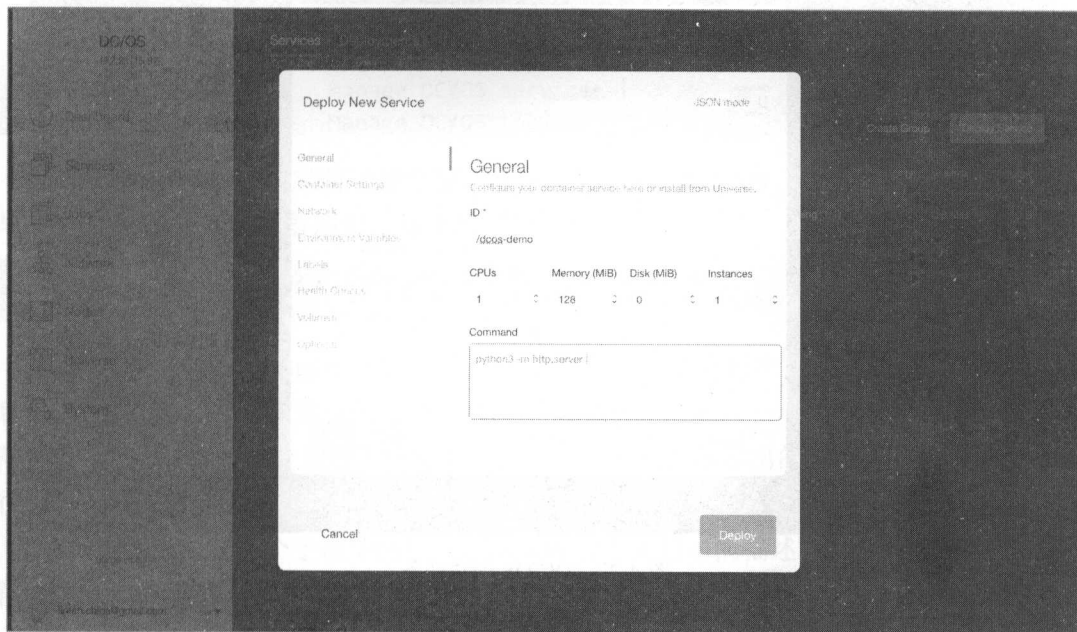


图 4-26 DC/OS 创建服务

Jobs 面板用于管理由 Metronome 创建的任务，其布局与 Services 面板一致。单击“New Job”按钮会打开创建任务窗口，窗口的右上角同样有“JSON mode”切换器，如图 4-27 所示。这里有个需要注意的地方，在任务的“Schedule”栏目中有一个“Run on a schedule”选项，默认是没有勾选的，此时创建的任务需要在界面上或通过 API 来手动触发执行。如果希望任务定时自动运行，应该勾选该选项，然后使用 Crontab 表达式定义任务的重复执行时间。

值得一提的是，DC/OS 并没有刻意隐藏其背后的各个功能组件，与之相反，它将所有的这些组件尽可能地开放给用户。例如，若想绕过 DC/OS 的接口直接访问数据中心的系统 Marathon 服务，只需将请求发到 Master 节点地址的/marathon/路径，访问“http://<Master 节点 IP>/marathon/”可以打开原生的 Marathon 界面，其他 API 地址可以此类推。相应地，原生 Mesos 界面路径是“http://<Master 节点 IP>/mesos/”。这个特性归功于 DC/OS 中的 AdminRouter^①组件，可以在该组件的 GitHub 页面上找到更多相关的信息。

^① <https://github.com/dcos/adminrouter>

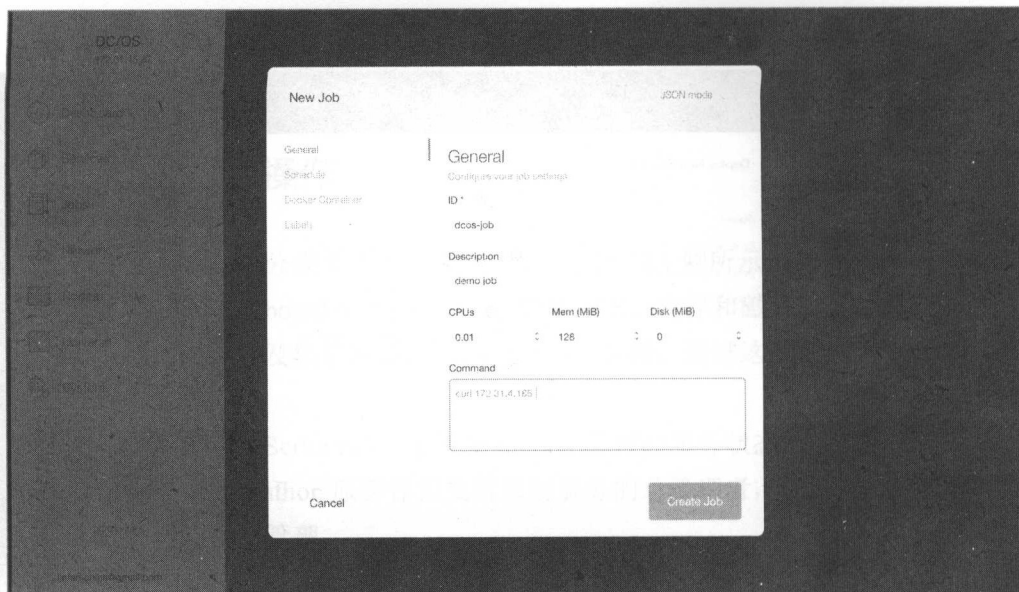


图 4-27 DC/OS 创建任务

4.6.4 DC/OS 命令行工具

前文介绍 Marathon 时已经介绍了如何安装和使用 `dcos` 命令行工具来简化 Marathon 的服务部署和管理。实际上，除了 `marathon` 子命令，`dcos` 命令行工具中的大部分功能都是专门针对 DC/OS 集群操作系统设计的，如下所示。

```
$ curl -O https://downloads.dcos.io/binaries/cli/linux/x86-64/dcos-1.8/dcos
$ chmod +x dcos
$ sudo mv dcos /usr/local/bin/
```

```
$ dcos
Command line utility for the Mesosphere Datacenter Operating
System (DC/OS). The Mesosphere DC/OS is a distributed operating
system built around Apache Mesos. This utility provides tools
for easy management of a DC/OS installation.
```

Available DC/OS commands:

<code>auth</code>	Authenticate to DC/OS cluster
<code>config</code>	Manage the DC/OS configuration file
<code>help</code>	Display help information about DC/OS

job	Deploy and manage jobs in DC/OS
marathon	Deploy and manage applications to DC/OS
node	Administer and manage DC/OS cluster nodes
package	Install and manage DC/OS software packages
service	Manage DC/OS services
task	Manage DC/OS tasks

使用 `dcos` 命令行的完整功能之前，需要先配置 `core.dcos_url` 参数并登录，使之能与正确的 DC/OS 目标集群通信，如下所示。

```
$ dcos config set core.dcos_url http://172.31.15.87
$ dcos auth login
```

输入 `dcos auth login` 命令后，`dcos` 命令行并不会让用户直接在控制台输入登录信息，而是打印出一个集群登录的 URL 地址，然后提示用户输入一个 OpenID 的 Token 序列。需要在浏览器中访问这个地址，完成登录操作。将界面上显示出的 Token 序列拷贝到命令行上，即可完成登录验证。

关于 `dcos` 工具各个子命令的使用，DC/OS 官方文档中的描述^①已经十分详细，这里不再对此工具的使用细节进一步展开，如有需要，读者可参考官方文档 CLI 部分的内容。

4.6.5 DC/OS 的应用仓库

DC/OS 提供了应用包管理服务，使用户可以像部署 Linux 中的 RPM/DEB 包那样在数据中心中部署应用。但与普通 Linux 系统不同的是，DC/OS 应用仓库提供了一种打包和交付大型分布式应用的方式。在 Web 界面上可以看到 DC/OS 使用了一个很霸气的名词来命名它的应用仓库：Universe（宇宙）。

实际上，Universe 项目仅仅是 DC/OS 应用仓库服务的名字，而真正完成与 Mesos 和 Marathon 通信以及创建服务等一系列操作的是另一个名字同样霸气的功能组件：Cosmos^②（宇宙的另一个同义词）。它在后台默默地维护着 DC/OS 中仓库列表的信息，并在 Web 界面上戴着 Universe 的面具完成软件包的查询、部署和删除工作。例如，当用户在系统配置页面添加了一个仓库地址时，后台调用的其实是 Cosmos 组件的 API。

那么真正的 Universe 服务究竟是做什么的呢？

直观来讲，Universe 有点像 Docker Hub，是在集群之外的一个单纯提供应用包信息的

① <https://dcos.io/docs/1.8/usage/cli/command-reference/>

② <https://github.com/dcos/cosmos>

仓库。不过在 Universe 中并没有像 Docker Hub 那样包含要部署的服务文件本身，而是仅仅包含了应用包的元数据。为了弄清楚个中细节，不妨看一下 Universe 的 GitHub 仓库^①，如下所示。

```
$ git clone https://github.com/mesosphere/universe
$ ls universe
docker  hooks  LICENSE  README.md  repo  scripts
```

仓库里应用包都在“repo”目录下，这个目录包含两个子目录。“meta”子目录存放的是整个 repo 的元数据，便于 Cosmos 对包进行检索，“packages”子目录存放的是每个应用包的元数据。在“packages”子目录中又以应用包的头字母命名了一系列子目录。比如在“M”目录中包含的是仓库中所有以字母“M”开头的应用包，如下所示。

```
$ tree universe/repo/packages/M/
universe/repo/packages/M/
├── marathon
│   ├── ...
│   └── 7
│       ├── config.json
│       ├── marathon.json.mustache
│       ├── package.json
│       └── resource.json
├── marathon-lb
│   ├── ...
│   └── 16
│       ├── config.json
│       ├── marathon.json.mustache
│       ├── package.json
│       └── resource.json
... ..
```

可以看到，在每个应用包中，又使用数字版本建立子级目录，用来做应用的多版本控制。在最末层的目录中，都包含了 4 个文件。它们便是应用的元数据了，在 Universe 的 GitHub 页面上有这 4 类文件的详细描述。

- package.json

package.json 文件提供应用包的大致介绍，包括应用包的版本、应用包名称、功能描述、维护者信息等，这些数据都会被 Cosmos 收集并且展示到 DC/OS 的 Web 界面里。

- config.json

^① <https://github.com/mesosphere/universe>

config.json 文件提供了应用部署时需要的参数,比如应用所需要创建的 Marathon 实例副本个数以及每个副本所需占用的 CPU 和内存的默认大小等。当用户通过 DC/OS 来部署应用的时候,这些参数值都会作为应用的高级配置展示给用户,用户可以根据实际需求在创建服务前修改它们。

- resource.json

resource.json 文件定义了应用在 DC/OS 上显示的图片、图标、提示信息 and 所需的容器镜像等内容。这些内容可以在 marathon.json.mustache 中引用,从而将具体数据与部署模板分离开。

- marathon.json.mustache

marathon.json.mustache 是用来创建 Marathon 应用的模板文件,其格式与 Marathon 的应用描述文件基本一致,但不包含与资源使用相关的配置,它们会在部署时从“config.json”文件以及用户的输入整合后,被加到模板里面。

在有些较早的仓库里还会有一个“command.json”文件,里面包含了一些额外的脚本信息,现在已经被“resource.json”文件替代。

了解了 Universe 仓库的结构和作用以后,用户们是否能够在 DC/OS 的应用仓库里增加一些自定义的应用包呢?在默认情况下,DC/OS 集群会从官方的应用仓库中获取应用列表,这个仓库中的内容是无法修改的。不过既然人家都把整个仓库的代码开源出来了,用户不妨自己动手搭建一个仓库,然后让 DC/OS 服务连接自己的仓库,这样就可以随意修改仓库里的应用了。

Universe 仓库是用 Python 3 编写的,在构建仓库前,需要预装 jq 命令和 Python 3 的“pip”工具。

对于 Ubuntu 系统来说,此步骤相对比较简单,如下所示。

```
$ sudo apt install -y jq python3-pip
```

若是 RHEL 和 CentOS 系统就要稍微麻烦一点,需要先安装 EPEL 和 IUS 的仓库源,如下所示。

```
$ sudo yum install -y https://dl.fedoraproject.org/pub/epel/
  epel-release-latest-7.noarch.rpm
$ sudo yum install -y https://centos7.iuscommunity.org/ius-release.rpm
$ sudo yum install -y jq python34u-pip
```

然后在 Universe 的 repo 仓库中依葫芦画瓢地创建自己的应用包元数据文件。这里不再给出具体实例,参照仓库中已有的应用包修改即可,注意应用的版本号是从数字 0 开始的。

接着在“universe”代码目录中执行 scripts/build.sh 和 docker/server/build.

bash 来完成对元数据文件的压缩打包和生成本地 Docker 镜像。可以将此镜像推送到镜像仓库，或直接在当前节点启动 Universe 仓库服务，如下所示。

```
$ pip3 install jsonschema
$ scripts/build.sh
$ docker/server/build.bash
$ docker run -d -p 8085:80 --name mesosphere-universe \
    mesosphere/universe-server:dev
```

最后替换 DC/OS 集群的应用仓库地址，可以在 Web 界面的系统配置页里更改，或者使用 dcos 工具，如下所示。

```
$ dcos package repo list
Universe: https://universe.mesosphere.com/repo
$ dcos package repo remove Universe
$ dcos package repo add --index=0 demo-universe http://172.31.4.187:8085/repo
$ dcos package repo list
demo-universe: http://172.31.4.187:8085/repo
```

然后再次查看 Web 界面的仓库包列表，就会看到自定义的应用已经加入到仓库中。

4.7 本章小结

作为数据计算领域十分成熟的知名调度器产品，Mesos 在与容器技术结合后形成了其独有的服务部署、调度和管理风格，可谓老树开新花。本章从 Mesos 的发展历史入手，向读者阐述了该项目的背景和“资源——任务”两层调度架构设计，然后详细讲解了其中的 Marathon 和 Chronos 两款任务调度框架，最后介绍了基于 Mesos 的开源数据中心操作系统产品 DC/OS。

相比于 SwarmKit 和 Kubernetes，基于 Mesos 的容器集群在部署和操作方面略为烦琐。但它能将现代化的容器服务与传统的非容器服务，特别是像 Hadoop 和 Spark 这样的大型数据计算框架进行统一的资源分配管理，充分利用现有硬件平台。

第5章 Rancher 集群解决方案

Rancher 是一套开箱即用的完整的容器编排解决方案，提供了包括网络、存储、路由、安全、服务发现和资源管理等十分丰富且易用的功能集合。它与前几个章节介绍的集群方案主要通过命令行工具操作的设计思路具有很大不同，Rancher 在最初的时候就非常重视 Web UI 的用户体验，并且已经支持中文界面，因此在使用难度上相对较低。除了自身可以管理容器以外，Rancher 还能够辅助用户快速部署 SwarmKit、Kubernetes 或 Mesos 的集群，并在 Web UI 中对这些集群运行的容器进行管理。本章将着重讲解基于 Rancher 内置的容器调度器 Cattle 的使用，最后简单介绍通过 Rancher 部署其他类型集群的方法。

5.1 Rancher 集群概述

5.1.1 Rancher 项目的起源

总部位于美国硅谷的 Rancher（图 5-1）的母公司 Rancher Labs 成立于 2014 年 9 月，它的创立者是被称为“技术梦想家”的梁胜。在创办 Rancher Labs 以前，梁胜博士曾经是 Sun Microsystems 公司核心的工程师，领导了 Java 虚拟机事实标准 Sun JVM 的设计。2008 年他创立 Cloud.com 公司，并推出开源的商业云计算基础设施即服务产品 CloudStack。2011 年 7 月，Citrix 公司收购 Cloud.com，梁博士出任 Citrix 云平台首席技术官，成为该公司的首位华人 CTO。这些经历使得 Rancher 在诞生后就得到了技术社区的高度关注。

除了 Docker 调度平台 Rancher，Rancher Labs 旗下还有一款极轻量的容器定制款 Linux 发行版：RancherOS。这款颇具特色的操作系统充分利用了容器的灵活性，实现了可插拔的核



图 5-1 Rancher 项目的 Logo

心组件设计，本书在第 8 章中会详细地介绍它。

2016 年 6 月，Rancher Labs 与国内的东网科技公司合资，正式成立了 Rancher Labs 在大中华区的分公司：源澈科技公司，总部位于深圳。同时上线了 Rancher 中文网站“cnrancher.com”^①。随着 Docker 应用的普及，本土优势使得 Rancher 在国内发展颇具潜力。目前，Rancher 项目在国内已经积累了可观的企业级案例，线上技术社区十分活跃，每年都有数场线下社区聚会。2017 年 9 月，Rancher 发布了全新的 2.0 技术预览版本，在操作界面上有较大调整。至本书截稿时，该系列依然处于 Alpha 阶段，本小节的内容将采用当前稳定的 1.x 版本为例，其大部分操作将同样适用于 2.0 之后的版本。

5.1.2 Rancher 的结构

从整体上看，Rancher 的集群同样使用了 Server-Agent 的主从部署结构，Server 节点提供集群管理的 API 并保存集群状态信息，Agent 节点执行实际的用户任务。Rancher 的所有服务都运行在容器中，这使得它从外部看起来是一个单独的整体，但实际却是由许多小服务构成的。这些服务的源码主要在 GitHub 上，以下是其中的一部分关键项目。

- github.com/rancher/rancher: 构建 Rancher 服务镜像的脚本和辅助工具。
- github.com/rancher/ui: 功能强大的 Web UI 管理界面。
- github.com/rancher/cattle: 容器运行管理器，它是整个 Rancher 的核心。
- github.com/rancher/scheduler: 辅助 Cattle 完成任务调度的子服务。
- github.com/rancher/healthcheck: 辅助 Cattle 完成任务健康检查的子服务。
- github.com/rancher/rancher-auth-service: 辅助 Cattle 管理用户权限和登录的子服务。
- github.com/rancher/rancher-dns: 辅助 Cattle 完成服务域名解析的子服务。
- github.com/rancher/rancher-net: 辅助 Cattle 完成网络管理功能的子服务。
- github.com/rancher/rancher-metadata: 辅助 Cattle 管理集群元数据信息的子服务。
- github.com/rancher/catalog-service: 辅助 Cattle 完成应用商店功能的子服务。
- github.com/rancher/rancher-catalog: 官方应用商店的服务模板列表。
- github.com/rancher/community-catalog: 社区贡献的应用商店服务模板列表。
- github.com/rancher/agent: Rancher 的 Agent 端服务。
- github.com/rancher/cli: Rancher 的用户命令行工具。

^① <http://www.cnrancher.com>

它们大致构成如图 5-2 所示的三层微服务结构。

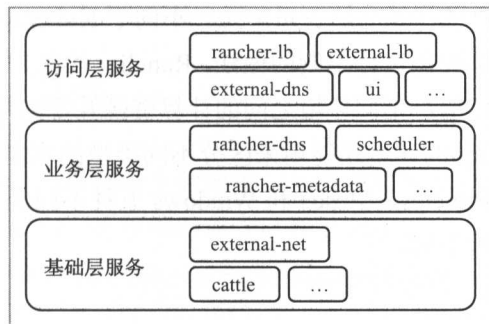


图 5-2 Rancher 服务的大致结构

在 Rancher 服务中，最核心的部分是 Cattle，它是一个可扩展的容器管理引擎。通过 Cattle 以及扩展服务，Rancher 实现了完整的容器即服务功能。实际上，Cattle 设计的独特之处在于其不但能够管理容器，还能管理其他的容器调度器，号称全球唯一兼容 SwarmKit、Kubernetes 和 Mesos 三种调度器的企业级容器管理平台。

此外，在 Rancher 中还有许多与网络、存储和安全等相关的特性通过插件的形式提供，确保了整体架构的灵活性。

5.1.3 相关概念

在 Rancher 中有一些特有的概念，它们组成了集群的整体结构。在开始使用 Rancher 之前，有必要先简单了解一下这些概念之间的关系，如图 5-3 所示。

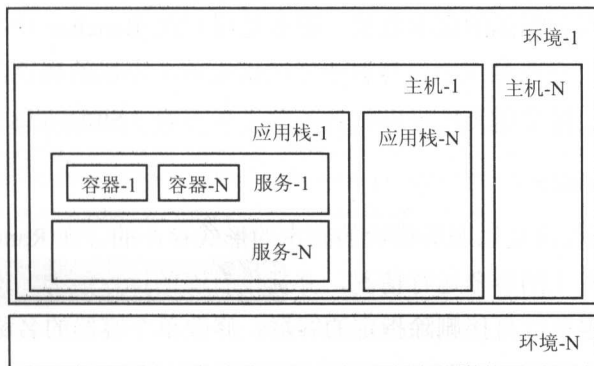


图 5-3 Rancher 中的重要概念关系

1. 环境 (Environment)

环境是 Rancher 对租户进行隔离的逻辑单元。不同于 Kubernetes 的 Namespace 让不同租户共享集群资源并对资源配额进行限制的做法，Rancher 的租户是在主机层面隔离的，每个主机能且只能属于一个环境，不同环境之间的计算资源互不共享。

除了隔离租户，Rancher 的环境还能用来区分不同类型的容器调度器。Rancher 目前支持 Cattle、SwarmKit、Kubernetes、Mesos 和 Windows 五种不同的环境类型，它们在界面、部署的组件和配置上均有所不同。

2. 主机 (Host)

主机即集群中运行了 Agent 服务的独立节点。Rancher Agent 服务会根据节点所属的环境类型，自动创建必要的辅助服务，例如网络服务、存储服务、调度服务等，它们都以容器的形式在主机上运行。有些 Rancher 插件也会在每个主机上运行相应的服务容器。

3. 应用栈 (Stack)

应用栈是多种不同容器的组合，它们共同协作提供某项业务功能。在同一个应用栈中的容器能够通过 Rancher 内置的 DNS 服务相互通过名称直接访问。

某种层面上说，应用栈与 Kubernetes 的 Pod 在某些方面有点相似，但 Kubernetes 的 Pod 是整体伸缩的，而 Rancher 应用栈中的容器可以单独伸缩。此外，Kubernetes 的同一个 Pod 的容器会被部署在同一个主机，而 Rancher 采用了一种“从容器 (Slidekicks)”的概念来解决这个问题，有关“从容器”的细节将在第 5.3.2 小节中介绍。

4. 服务 (Service)

在 Rancher 中，服务实际上指的是容器的副本模板，它保存有创建一个服务容器所需的所有配置信息以及当前容器的副本数量。服务是用户在 Rancher 中所能管理的最小业务单元，也就是说，用户不能直接创建容器或修改指定单个容器的属性，必须借助相应的服务单元来操作。此外，每个服务还可以包含多个“从容器 (Slidekicks)”服务。

5. 容器 (Container)

Rancher 中的容器仅仅是以服务模板的副本的形式存在的。在 Rancher 的 Web UI 上能够直观地看到各个主机上的容器运行情况，并提供有限的操作能力，例如查看容器运行状态和资源使用情况，也可以直接删除指定的容器、修改单个容器的名称和端口映射等。

5.2 构建 Rancher 集群

5.2.1 部署 Server 节点

Rancher 集群的 Server 节点部署十分简单。只需要使用一个安装了 Docker 的服务器，然后运行一条启动 Rancher Server 容器的命令，如下所示。

```
$ docker run -d --restart=unless-stopped -p 8080:8080 \
  --name rancher rancher/server:latest
```

这个容器包含了启动 Rancher 服务所需的全部环境依赖。第一次运行时，Rancher 会创建一些基本的数据文件，所需时间根据节点的配置和网络情况不同，大概需要 2~5 分钟。可以使用 `docker logs` 命令观察它的初始化过程，如下所示。

```
$ docker logs -f rancher
```

直到日志中出现 “Starting websocket proxy. Listening on [:8080]” 这样的内容，就表明初始化基本完成了。

此时用浏览器打开 “<http://<服务器 IP 地址>:8080>” 页面，就会看到 Rancher 的管理页面了，如图 5-4 所示。

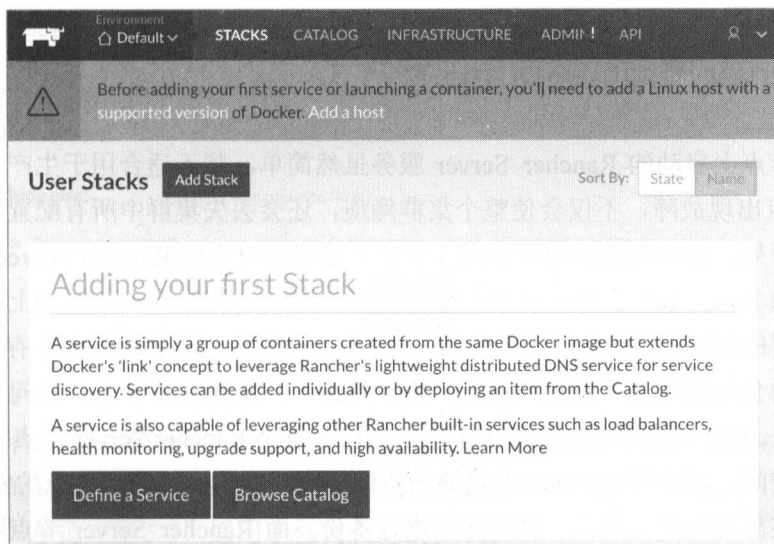


图 5-4 首次进入 Rancher 时的管理页面

此时在顶部“ADMIN”菜单的右边有一个红色的叹号，这是因为还没有为 Web UI 配置用户验证。单击这个红点就会进入访问控制的配置页面。Rancher 集成了多种登录方式，包括适合企业使用的 Windows Active Directory 和 LDAP，以及适合社区使用的 GitHub 等，如图 5-5 所示。

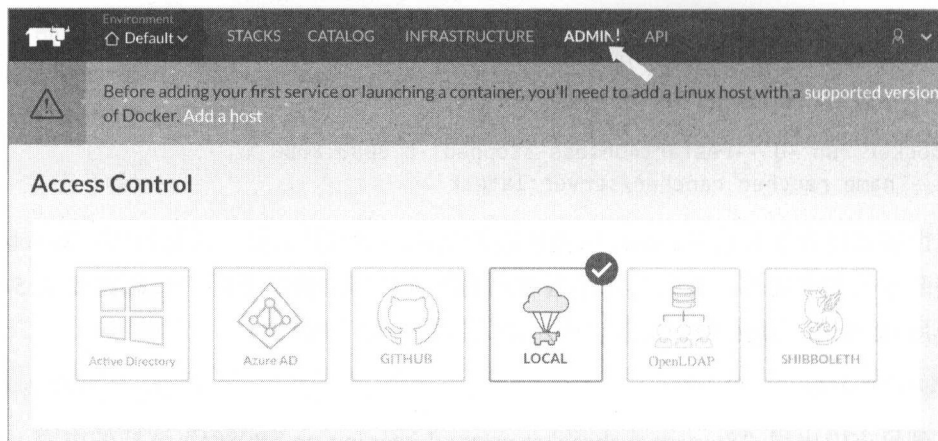


图 5-5 Rancher 的访问控制页面

如果不想使用任何现有的账号集成，Rancher 也允许用户使用普通的用户名密码方式登录。只需选择“LOCAL”这项，然后在下面的配置里输入管理员的用户名和密码。

5.2.2 Server 节点的高可用部署方式

在单个节点上启动的 Rancher Server 服务虽然简单，却不适合用于生产环境。一旦这个 Server 节点出现故障，不仅会使整个集群瘫痪，还会丢失集群中所有配置的数据，造成无法恢复的后果。此时应该采用至少由 3 个节点组成高可用的 Rancher Server 集群。

高可用的 Rancher 除了 Server 节点数量更多以外，为使各 Server 节点上的数据一致，还应部署外置的 MySQL 服务，让所有 Rancher Server 都通过外置的数据库存储数据。并添加一个额外的负载均衡器（比如 Nginx）来让外部用户能够从单一入口访问集群中的当前可用的主节点。对于规模比较大的正式环境，建议为每个 Rancher Server 节点提供至少 8GB 以上的内存空间，以确保 Rancher 有足够的资源可用。为了确保数据信息的安全，MySQL 服务器也应该部署为主从模式，并适时地进行备份。而 Rancher Server 节点除了开放用于访问界面和 API 的 TCP/8080 端口，还应该开放用于节点间数据通信的 TCP/9345 端口。

准备就绪后，在每个作为 Rancher Server 的节点上执行以下命令。

```
$ docker run -d --restart=unless-stopped \
  -p 8080:8080 -p 9345:9345 \
  rancher/server:latest \
    --db-host myhost.example.com --db-port 3306 \
    --db-user <数据库用户名> --db-pass <数据库密码> \
    --db-name cattle \
    --advertise-address <当前节点的 IP 地址>
```

等待所有节点上的服务启动完成，通过外部的负载均衡就可以随时访问到正常运行的 Rancher Server 了。

Rancher 的 Web UI 默认使用 HTTP 协议提供服务，在正式的对外环境里，还应该配置 HTTPS 验证。通常只需在负载均衡器上配置 SSL 证书，并未特别之处，在 Rancher 文档中提供了一些常用的负载均衡工具的 HTTPS 配置^①，可作为参考。

5.2.3 添加 Agent 节点

进入 Rancher 界面，在顶部菜单栏选择“Infrastructure”，再从下拉菜单中选择“Hosts”，在该页单击“Add Host”。第一次进入该页面时需要确认 Server 的 API 地址，如果默认识别到的地址在 Agent 节点所在的网络不可用的话，需要填入一个自定义地址，否则直接单击“Save”即可，如图 5-6 所示。

Host Registration URL

What base URL should hosts use to connect to the Rancher API?

☒ **This site's address:**

☐ **Something else:**

Don't include /v1 or any other path, but if you are doing **SSL termination** in front of Rancher, be sure to use **https://**

Save

图 5-6 确认 Rancher API 访问地址

Rancher 内置了 AWS EC2、Azure 和 DigitalOcean 的 Agent 自动安装模块（可以通过插件添加更多节点类型），用于批量添加和初始化 Agent 节点。而对于少量节点的添加，第一种“Custom”方式（图 5-7）适用于任何类型的主机。

^① <http://docs.rancher.com/rancher/v1.5/en/installing-rancher/installing-server/basic-ssl-config/>

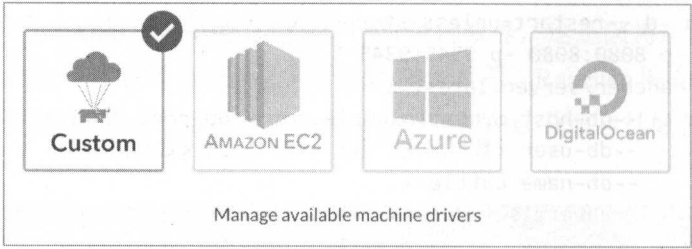


图 5-7 选择主机类型

选择“Custom”部署方式，在页面下方会看到一段 docker 命令，拷贝这个命令，在所有作为 Agent 的节点上（预装 Docker）执行一次，如下所示。

```
$ sudo docker run -d --privileged \
-v /var/run/docker.sock:/var/run/docker.sock \
-v /var/lib/rancher:/var/lib/rancher \
rancher/agent:v1.2.1 \
http://x.x.x.x:8080/v1/scripts/92D0B595DB20D5.....kWuv7QD8YPujrIc
```

等待几分钟，回到 Web UI 的“Infrastructure→Hosts”页面，就会看到新的节点已经添加进来了，如图 5-8 所示。

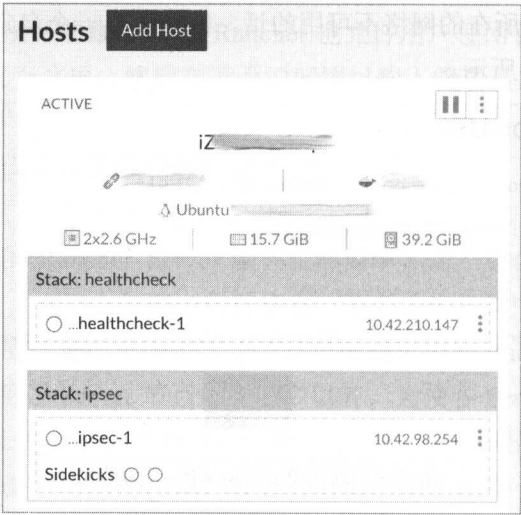


图 5-8 新增加的 Agent 节点

在这个页面里，Rancher 列出了每个节点上当前运行的所有容器（包括已经停止的容器），并以应用栈区分出来。可以看到在启动 Rancher Agent 容器后，这个容器自动创建了许多辅助的服务容器，分别运行在各个独立的应用栈里。在每个节点的最下方有一个标记

为“Standalone Containers”的特殊区域，这个区域中的容器通过 Docker 直接启动，没有使用 Rancher 的服务模型，但 Rancher 同样会将它们管理起来。

容器前面有颜色的小点表示该容器当前的状态，空心的绿色圆点表示容器正在运行，实心的红色圆点表示容器已经停止。有时候还会见到黄色叹号的三角形标识（表示相应容器已启动，但相应的健康检查没有通过），或者红色叹号的三角形标识（表示容器启动出错），此时就有必要检查容器运行不正常的原因了。

随意单击一个容器的名称即可进入该容器的资源使用情况和配置信息的页面。如果容器是通过 Rancher 创建的，还能跳转到相应的服务或应用栈页面里查看与容器运行配置相关的更多信息。

5.3 Rancher 的服务管理

5.3.1 使用 Rancher Web UI 创建服务

正如大家在部署 Agent 节点过程中看到的，Rancher 的绝大多数功能都可以通过界面完成。其实还有一个更大的惊喜，Rancher 的 Web UI 是内置中文支持的（咳……是不是觉得，怎么不早说）。不知道读者们是否注意到了，在页面右下角很不起眼的地方有一个“English”按钮，单击它，在弹出的选项中选择“简体中文”即可切换成中文显示了，如图 5-9 所示。



图 5-9 切换界面语言

所以，从这个小节开始的界面截图就统一使用中文界面了。

与部署服务相关的功能在“应用栈管理”页面，可单击首页顶端的“应用栈”标签进入，此时集群中还没有任何应用栈。单击最上方的“添加应用栈”按钮，进入创建应用栈的页面，如图 5-10 所示。

添加应用栈

名称

Demo

描述

Demo应用

可选:导入COMPOSE

可选:docker-compose.yml

docker-compose.yml文件的内容

上传

可选:rancher-compose.yml

rancher compose.yml文件的内容

上传

高级选项 ^

创建

取消

图 5-10 创建应用栈

在页面填入应用栈的名称和描述，下方有两个可选的文件上传选项，这是一种快速定义应用栈的方式，关于这两个文件的细节，将在第 5.3.5 小节中介绍。现在先单击“创建”按钮完成一个空应用栈的创建。

在应用栈详情页面里，可以看到当前这个应用栈里还没有任何服务，单击上方的“添加服务”按钮，进入创建服务的页面，如图 5-11 所示。

Add Service

数量

Run 2 个容器

总是在每台主机上运行一个此容器的实例

Demo

+ 添加从容器

名称

Demo

描述

Demo服务

选择镜像*

nginx:1.11-alpine

创建前总是拉取镜像

端口映射

图 5-11 创建服务

在页面中填入服务的副本个数、名称、描述等信息，然后提供一个容器镜像，这里以 Nginx 镜像为例，因为它内置了一个可以展示的欢迎页面。在选择镜像的位置旁有一个“创建前总是拉取镜像”选项，如果勾选它，则不论该容器是否已经有当前节点存在，都会重新拉取一次，可以根据实际情况决定是否需要勾选。

如果想不通过额外的负载均衡等手段直接访问创建的 Nginx 服务，可以单击“端口映射”前面的小加号按钮，添加一个容器到所在主机的端口映射，例如将容器 80 端口映射为主机的 8000 端口。最后单击页面底端的“创建”按钮。

创建完成后，在应用栈 Demo 的详情页面里，就会看到一个状态是“Activating”的服务了。等待片刻，当服务状态变为“Running”以后，单击该服务属性中带链接的“8000”端口位置，就会在新页面里看到一个已经部署好的 Nginx 欢迎页。

此时一个简单的服务就创建完成了。

5.3.2 从容器

观察在主机页面中运行的服务，会发现其中有一部分服务还带着一个或几个附属的容器，如 Agent 上的“ipsec”应用栈中的“ipsec”服务就包含两个附属的容器，如图 5-12 所示。它们在 Rancher 中被称为“从容器（Sidekicks）”。

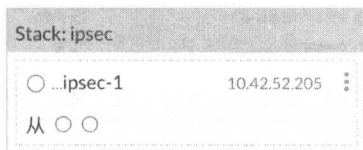


图 5-12 基础设施服务中的从容器

从容器具有一些独特的性质。例如从容器会随着所属的服务的扩缩而自动扩缩，始终保持与主服务相同的副本数。此外，Rancher 会确保从容器与所属的服务副本容器运行在同一个主机上，这是一个十分重要的特性，它弥补了 Rancher 的原始的服务模型中缺少“群组调度（Gang Scheduling）”能力的短板。

几个容器始终运行在同一个节点上，意味着这些容器间的网络通信成本最低，且它们可以通过本地文件挂载在共享存储。这使得用户得以将一些关系紧密的服务部署在一起，让每个服务各司其职，而又相互配合做好一件事。例如，一个服务采集信息，另一个服务对采集到的信息进行处理。从容器有一种很常见的应用场景。就是用来储存主服务所需要持久化的数据。这样既能避免数据直接暴露在主机上，又能使得服务在升级过程中不会丢失这些数据，在应用商店中的很多服务都使用了这样的设计。

下面来实际演示一下。

首先在 Demo 应用栈里新建一个名为“DemoMain”的服务，使用一个较低版本的镜像，例如“nginx:1.10-alpine”镜像。然后在创建服务页上方单击“添加从容器”按钮，添加一个名为“DemoSidekick”的从容器，如图 5-13 所示。



图 5-13 创建从容器

打开 DemoSidekick 容器最下方的“命令”标签，在其中的“命令”选项中写入“true”，在“自动重启”选项中选择“从不（仅启动一次）”，如图 5-14 所示。从容器可以使用任何镜像，例如“alpine:latest”，因为仅将它作为数据存储来使用，通常会选择一个启动后就立即结束的镜像，以节约资源开销。



图 5-14 修改从容器的命令选项

在从容器的“卷”标签里添加主服务要持久化保存的目录，例如“/var/log/nginx”，然后在主服务的“卷”标签里勾选挂载 DemoSidekick 从容器里的所有卷，如图 5-15 和图 5-16 所示。



图 5-15 从容器的卷配置



图 5-16 主服务的卷配置

这样一来，主服务要保存文件的目录实际上就被放在了从容器里，看起来似乎多此一举。但接下来就要对服务进行替换镜像的升级。升级前先进入主服务的容器，往“/var/log/nginx”目录中随意写入一些内容。回到服务的页面，在服务的菜单里选择“升级”，如图 5-17 所示。然后在升级的页面里将镜像修改成“nginx:1.11-alpine”。

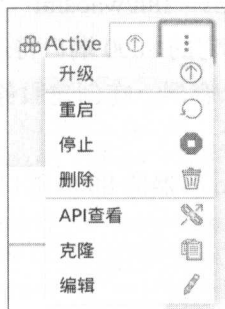


图 5-17 服务菜单中的升级功能

等待 Rancher 停掉当前版本的服务容器，然后重新启动一个新版本的服务容器，进入新容器的“/var/log/nginx”目录里，会发现之前写到这个目录里的文件还存在。通过这种方式，就将数据本身和它所属的服务解耦了。

关于服务的升级会在第 5.3.6 小节里详细介绍。

5.3.3 特殊类型的服务

在 Rancher 中，除了通过用指定容器创建的服务，还有三种特殊类型的服务，分别是负载均衡、服务别名、外部服务。在应用栈详情页面的“添加服务”按钮右侧，有一个向下的小箭头，单击这个箭头就会弹出一个新菜单，如图 5-18 所示。

下面将依次讲解这三种服务。

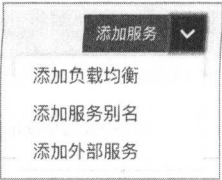


图 5-18 创建特殊类型的服务

1. 负载均衡

负载均衡服务用于将多个内部服务对外暴露成统一入口。在 Rancher 中，每个负载均衡服务实例能够处理属于同一应用栈中的其他多种服务请求，只需添加多个端口映射。在同一个负载均衡服务处理的属于相同服务的多个容器，会轮流收到用户的请求。

为了比较好地验收这种负载均衡效果，可以将 Demo 服务的镜像改为“flin/whoami”，并将容器的副本数量添加到 2 或以上。“flin/whoami”这个镜像在第 4.3.7 小节关于 Mesos 负载均衡的例子中已经介绍过，访问它的 8000 端口将返回当前容器内的主机名，通过这个信息就能知道当前处理请求的是哪个容器。在创建服务时，不必为它添加端口映射，因为接下来将使用负载均衡服务来访问它。

创建一个主机端口 8001 到 Demo 容器的 8000 端口的负载均衡规则，如图 5-19 所示。



图 5-19 创建负载均衡服务

完成后，单击界面中的服务列表中 DemoLB 服务后面的“8001”文字链接，或直接访问相应节点的 8001 端口，就能在新打开的窗口中看到一串字符，这其实是所访问容器的主

机名。刷新窗口后会看到这个字符串在几个值之间轮换变化，这证明负载均衡是有效的。

Rancher 的负载均衡功能是如何实现的呢？在服务列表页面单击 DemoLB 服务，进入这个服务的详细信息页面，可以发现这个服务所使用的镜像名称是 “lb-service-haproxy”。事实上，在创建负载均衡的时候，Rancher 创建了一个基于 HAProxy 定制的后台服务，它与第 4 章中介绍的 Marathon-LB 在原理上如出一辙。当负载均衡服务所绑定的用户服务的容器副本数量发生改变时，负载均衡服务会自动发现这个变化，并相应地更新路由规则，从而使外部访问请求始终能够连接到正确的后端容器。

2. 服务别名

在 Rancher 中，属于同一个应用栈的服务之间，是可以通过服务的名称直接相互访问的。例如前文在 Demo 这个应用栈中创建了 Demo、DemoMain 和 DemoLB 这三个单独的服务。在 Demo 服务所属的容器里，就可以直接用 DemoMain、DemoLB 这样的名称访问其他两个服务，如下所示。

```
$ docker exec r-Demo-Demo-1-4e1810cf ping DemoMain
PING DemoMain (10.42.145.93): 56 data bytes
64 bytes from 10.42.145.93: seq=0 ttl=64 time=0.062 ms

$ docker exec r-Demo-Demo-1-4e1810cf ping DemoLB
PING DemoLB (10.42.84.123): 56 data bytes
64 bytes from 10.42.84.123: seq=0 ttl=64 time=0.087 ms
```

如果试图用名称直接访问属于另一个应用栈的服务，如下所示。则是不能够联通的。此时需要使用完整的 Rancher 内网域名，在第 5.4.5 小节介绍 Rancher DNS 时，会进行详细介绍。

```
$ docker exec r-Demo-Demo-1-4e1810cf ping ipsec
ping: bad address 'ipsec'
```

有的时候需要给一个服务起多个名字。从实现上来说，也就是需要在 DNS 里插入一些额外的记录。例如，可以给 DemoLB 服务起一个别名，“LoadBalancer”，如图 5-20 所示。

然后在同一个应用栈的服务所属容器里访问这个别名，如下所示。

```
$ docker exec r-Demo-Demo-1-4e1810cf ping LoadBalancer
PING LoadBalancer (10.42.84.123): 56 data bytes
64 bytes from 10.42.84.123: seq=0 ttl=64 time=0.087 ms
```

实际得到的 IP 地址与 DemoLB 服务相同。

添加服务别名 ②

名称	描述
LoadBalancer	DemoLB的别名

目标

➕ 添加服务

DemoLB

⌵

创建

取消

图 5-20 添加服务别名

3. 外部服务

服务别名使得用户可以向 Rancher 的 DNS 中添加新的域名记录。但这仅限于在应用栈中已经存在的服务。而外部服务则为用户提供了一种能够向 DNS 中添加任意外部 IP 映射的功能。

例如可以将 Rancher Server 的地址添加到 DNS 中，将其命名为“rancher-api”，如图 5-21 所示。

添加外部服务 ②

名称	描述
rancher-api	Rancher Server的服务地址

目标

☒ 指向IP地址

☐ 指向主机名

➕ 添加目标IP

13.124.47.69

⌵

图 5-21 将 Rancher 服务地址添加为外部服务

然后访问这个外部名称的 8080 端口，如下所示。

```
$ docker exec r-Demo-Demo-1-4e1810cf wget rancher-api:8080
Connecting to rancher-api:8080 (13.124.47.69:8080)
wget: server returned error: HTTP/1.1 401 Unauthorized
```

由于在 Rancher 设置了用户验证，访问返回了 401 错误，但是可以看到这个域名解析已经成功了。如果需要通过 HTTP 方式访问 Rancher 的 API，还需要为它生成相应的 API

Key, 这第 5.5.1 小节中将介绍到。

5.3.4 使用应用商店

Rancher 的应用商店 (Catalog) 使得用户能够快速找到许多现成的应用, 然后一键部署到集群里。应用商店按照不同的应用仓库将应用分类。默认情况下, Rancher 已经内置了两个应用仓库: Library 和 Community。如图 5-22 所示。

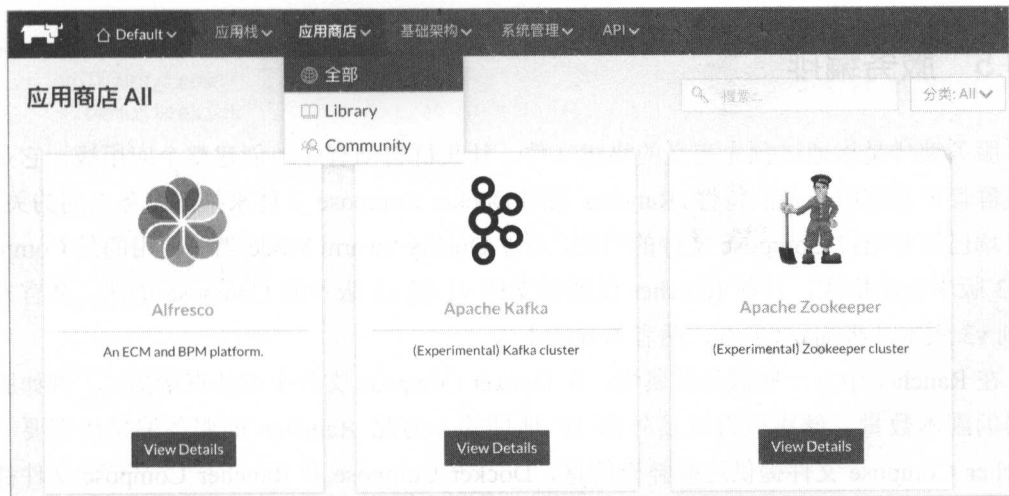


图 5-22 Rancher 的应用商店

Library 仓库中的内容来自 GitHub 上的 “rancher-catalog” 仓库, 其中的内容大多是官方提供的扩展插件服务。例如在这个仓库中有一个叫 “Rancher Container Cron” 的应用。单击该应用下方的 “View Detail”, 然后单击 “启动”。这样就为 Rancher 的集群增加了运行定时任务的功能。

在集群的任意节点里, 运行以下命令。

```
$ docker run -d -v /tmp:/tmp --label=cron.schedule="*/2 * * * * ?" \
  --name=date alpine sh -c 'date >>/tmp/date'
```

这个命令启动了一个带有 “cron.schedule” 标签的容器, 标签的值是一个标准的 Cron 定时表达式, 表示每隔 2 秒执行一次。现在将系统的临时目录挂载到容器里, 然后让容器往这个目录的 “data” 文件里追加写入当前的时间。过一会儿就会看到在临时目录的 “date” 文件里已经写入了许多时间记录。

Community 仓库中的内容来自 GitHub 上的“community-catalog”仓库，其中的应用大多是由社区贡献和维护的各类第三方软件，可以用顶部的搜索栏来快速查找仓库中的应用。以部署 Dokuwiki 为例，首先找到这个应用，单击“View Detail”，然后单击“启动”。几分钟后，一个 Dokuwiki 服务便部署好了。

不过，目前在 Rancher 的 Community 仓库中的服务版本大多已经较为过时，并没有什么实用性。在第 5.4.8 小节里会介绍如何创建自己的应用仓库，这样就能制作自己的定制版本应用了。

5.3.5 服务编排

服务编排是指通过预先定义的描述文件，让集群自动有序地创建整个应用栈，它是容器集群管理系统所必备的特性。Rancher 采用 Docker Compose 文件来描述服务之间的关系。第 2 章已经介绍过 Compose 文件的写法，不过 Docker Swarm Mode 当时使用的是 Compose 的 v3 版本语法格式，目前 Rancher 仅能够支持 v1 和 v2 版本的 Compose 语法，从官方的 v2 到 v3 版本升级描述^①来看，两者差异并不大。

在 Rancher 中有一些额外的属性，在 Docker Compose 文件中无法直接描述，例如服务容器的副本数量、健康检查以及外部 IP 地址等。为此 Rancher 的服务编排还需要一个 Rancher Compose 文件提供这些补充信息。Docker Compose 和 Rancher Compose 文件都采用 YAML 格式，通常使用“docker-compose.yml”和“rancher-compose.yml”命名，在两个文件的第一行都需要标注文件格式的版本，当前推荐使用的版本都是‘2’。

创建应用栈时，在如图 5-10 所示的对话框中，有两个可选的上传文件选项，用户可以在这里提供预先准备的这两个描述文件，这样当应用栈创建出来后会一步到位地自动创建用户所描述的所有服务。

以目前为止的 Demo 应用栈中的内容为例，若将它写成描述文件，可表示为下面这样。首先是“docker-compose.yml”文件，它包含了服务的主要信息，如下所示。

```
$ cat docker-compose.yml
version: '2'
services:
  Demo:
    image: flin/whoami
    stdin_open: true
```

^① <https://docs.docker.com/compose/compose-file/compose-versioning/#upgrading>

```

tty: true
DemoLB:
  image: rancher/lb-service-haproxy:v0.6.2
  ports:
    - 8001:8001/tcp
  labels:
    io.rancher.container.agent.role: environmentAdmin
    io.rancher.container.create_agent: 'true'
DemoMain:
  image: nginx:1.11-alpine
  stdin_open: true
  tty: true
  volumes_from:
    - DemoSidekick
  labels:
    io.rancher.sidekicks: DemoSidekick
DemoSidekick:
  image: alpine
  stdin_open: true
  volumes:
    - /var/log/nginx
  tty: true
  command:
    - 'true'
  labels:
    io.rancher.container.start_once: 'true'
Loadbalancer:
  image: rancher/dns-service
  links:
    - DemoLB:DemoLB
rancher-api:
  image: rancher/external-service

```

相应的“rancher-compose.yml”文件补充了 Rancher 特有的属性，它的格局与“docker-compose.yml”文件很相似，如下所示。

```

$ cat rancher-compose.yml
version: '2'
services:
  Demo:
    scale: 2
    start_on_create: true
  DemoLB:
    scale: 1
    start_on_create: true

```

```
lb_config:
  certs: []
  port_rules:
    - hostname: ''
      path: ''
      priority: 1
      protocol: http
      service: Demo
      source_port: 8001
      target_port: 8000
  health_check:
    response_timeout: 2000
    healthy_threshold: 2
    port: 42
    unhealthy_threshold: 3
    interval: 2000
DemoMain:
  scale: 1
  start_on_create: true
DemoSidekick:
  scale: 1
  start_on_create: true
Loadbalancer:
  start_on_create: true
rancher-api:
  external_ips:
    - 13.124.47.69
  start_on_create: true
```

在应用栈的 Web UI 里，Rancher 提供了将当前应用栈描述为 YAML 格式的功能，可见 YAML 文件与任意结构的应用栈都是可以对应上的，如图 5-23 所示。

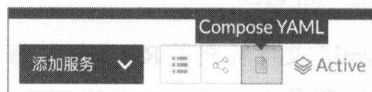
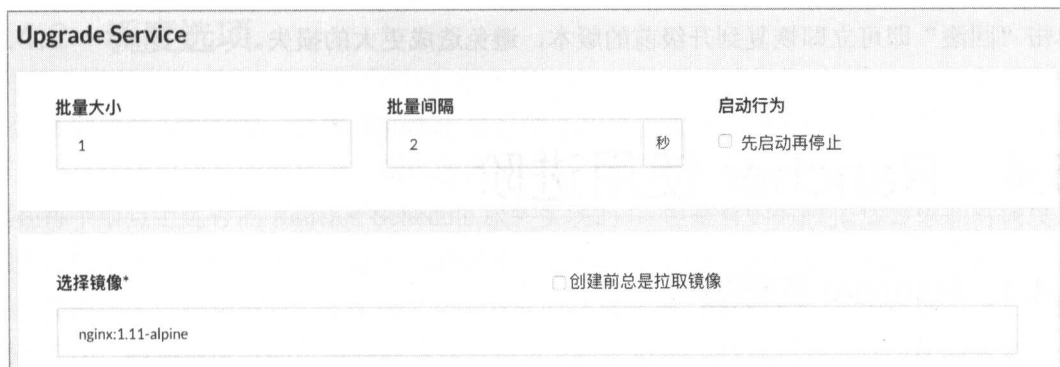


图 5-23 以 YAML 格式查看应用栈

5.3.6 服务的升级和回滚

在介绍从容器的时候，已经简单地提到了 Rancher 的服务升级。在 Web UI 上，可以通过服务菜单中的“升级”按钮进入服务升级页面，如图 5-24 所示。



The image shows the 'Upgrade Service' configuration page in Rancher. It includes three input fields: '批量大小' (Batch Size) set to 1, '批量间隔' (Batch Interval) set to 2 seconds, and '启动行为' (Startup Behavior) with the checkbox '先启动再停止' (Start first, then stop) selected. Below these is a '选择镜像*' (Select Image) section with a dropdown menu showing 'nginx:1.11-alpine' and a checkbox '创建前总是拉取镜像' (Always pull image before creating) which is unchecked.

图 5-24 服务升级的页面

服务的升级通常是指对服务镜像的修改，特别是更换镜像版本。同时也可以对端口映射、命令、卷等配置同时进行更新。和其他的平台一样，在 Rancher 进行服务的升级，事实上就是将旧的容器停止，然后重启相应数量的新的容器。

Rancher 采用的服务升级方式也是滚动升级，通过调整此页面最上方的“批量大小”和“批量间隔”两项参数，能够改变在滚动升级时，每批升级容器的个数和每个批次之间间隔多长时间（以便新容器中的服务器启动完成，避免服务中断）。最右侧的“先启动再停止”是一项十分有用的功能，在没有勾选的情况下，新版本容器的启动和旧版本容器的停止是同时进行的，这有可能导致新的容器还没有启动完成，旧的容器就已经停止了，造成服务中断。勾选这个选项可以避免这种情况发生。

当新的容器全部启动完成之后，Rancher 并不会立即删除旧的容器，此时服务处于一个更新结果待确认的状态。服务的菜单中会增加“升级完成”和“回滚”两个子项，如图 5-25 所示。



图 5-25 服务升级确认和回滚

用户可以先验证更新后的结果是否与预期一致，如果一切正常，则可以通过单击“升级完成”确认这次升级，此时 Rancher 会删掉旧版本的容器。如果发现新版本运行有问题，

单击“回滚”即可立即恢复到升级前的版本，避免造成更大的损失。

5.4 Rancher 使用进阶

5.4.1 Rancher 的标签

在 Rancher 中，许多有趣的功能是通过容器标签来实现的，例如前面介绍过用“cron.schedule”标签实现容器的定时运行。

另一个比较实用的场景是通过标签让普通 Docker 容器加入 Rancher 的跨节点网络。

如果注意观察在主机页面里的 Standalone Containers 容器和在应用栈中的容器，会发现它们的 IP 地址完全不在同一网段下。在主机上创建的容器会从主机的 docker0 网桥获得一个节点内唯一的 IP 地址，通常是 192.168.x.x 的网段，这会导致在不同节点上的 IP 地址重复。Docker 通过创建自定义 Overlay Network 可以让容器获得跨节点通信的能力，但早在这以前，Rancher 就已经为自己的集群内置了跨节点网络（默认是 IPsec 实现，可以通过插件替换为 VxLAN）。通过 Rancher 创建的容器是原生支持跨节点通信的，相应容器获得的 IP 地址是 10.x.x.x 网段。可以通过在集群中任意节点上，进入一个由 Rancher 创建的容器，然后 ping 运行在另一个节点上的 Rancher 托管容器的 IP 地址进行验证。

如果要想通过 Docker 直接启动的容器也纳入 Rancher 的 Overlay 网络，只需要在启动容器时加上 `io.rancher.container.network=true` 标签就可以了，如下所示。

```
$ docker run -d --name standalone \
  --label io.rancher.container.network=true nginx:1.11-alpine
```

回到 Rancher 的主机页面上观察这个新启动的容器，虽然还是被归在 Standalone Containers 类别里，但它的 IP 地址已经是 Rancher Overlay 网络的网段了，尝试在其他节点的 Rancher 容器访问它，也是可以联通的。

类似的标签还有让容器能够指定容器 IP 地址的 `io.rancher.container.requested_ip`、让容器能够使用 Rancher DNS 服务的 `io.rancher.container.dns` 以及让容器主机名自动替换成容器名称的 `io.rancher.container.hostname_override` 等。这里就不逐一介绍了，有兴趣的读者可以参考 Rancher 文档有关 Labels 的章节^①。

^① <https://docs.rancher.com/rancher/v1.5/en/cattle/labels/>

5.4.2 调度选项

在创建服务的时候，虽然 Rancher 会自动选择最合适运行的节点（主要是端口冲突检查以及节点可用计算资源检查，确保服务所需的端口在目标主机可用且资源足够，这是通过在每个节点上运行的 Rancher Schedule 服务实现的），但是有的时候依然希望能对调度的结果进行更加定制化的干预。这个功能同样可以通过给资源添加标签的方式来完成。

在界面上创建服务时，会看到有一个调度的控制版块。Rancher 提供两种用户自定义调度的方法：将当前服务的所有容器运行在指定主机上或基于特定规则选择匹配的主机运行容器。

若选择前一种方式，一旦用户所指定的主机出现故障，则相应的服务也将不可用，因此用这种方法部署的服务并不是高可用的。

对于后一种方式，Rancher 能够使用的规则包括主机标签、容器标签、服务名称及容器名称，如图 5-26 所示。

图 5-26 服务部署的调度选项

其含义如下所示。

- 主机标签：服务只在有特定标签的主机上运行。
- 容器标签：服务只在已经运行有特定标签容器的主机上运行。
- 服务名称：服务只在已经运行有特定名称服务的主机上运行。
- 容器名称：服务只在已经运行有特定名称容器的主机上运行。

通常来说，后三种约束意味着该服务与另一个特定的服务有紧密的关联关系，只是指定这个服务的方式不同罢了。

这里稍微介绍一下主机的标签和容器的标签。它们都属于相应资源的元数据，且本质

上都是键值字符串。主机的标签可以通过 Rancher Web UI 的主机页面选择特定主机后，在下拉菜单单击“编辑”，然后在编辑页面里添加或修改。容器的标签通常是在创建服务的时候加上的，在服务升级时可以修改服务标签，服务标签配置修改后，相应的容器会自动获得这些标签。

对于每条规则，用户后续需要从“必须”“必须没有”“最好”和“最好没有”中指定一种匹配条件，前两种表示如果相应规则不满足，则服务的容器将调度失败，后两种表示相应规则不满足，则服务将随机调度。此外可以为服务添加多条约束规则，只有同时满足所有规则的主机才会运行这个服务的容器。

除了调度运行指定个数的容器，Rancher 还支持一种特殊的全局服务，也就是说在集群的所有节点上都运行指定的服务，这对于某些监控或日志收集类的服务十分有用。它的配置在服务创建或更新页面的最上方，也就是如图 5-19 所示的修改容器副本数目的选项下方。

5.4.3 服务健康检查

除了调度和网络相关的服务，Rancher 在各个 Agent 节点上还运行有一个专门的健康检查（healthcheck）服务，如图 5-27 所示。它们实现了对服务容器的分布式健康检查。



图 5-27 Rancher Agent 上的基础设施服务

当创建服务时，用户可以在健康检查的面板里配置基于 TCP 或 HTTP 方式的健康检查规则，如图 5-28 和图 5-29 所示。

当服务的健康检查开启后，服务的每个容器都会被随机分配的 3 个不同节点定时访问检查（如果集群的 Agent 总数不到 3 个，则使用实际 Agent 的数目），每次检查只要有任意一个 Agent 上的 healthcheck 服务返回“检查通过”，容器就会被认为是健康的，只有当所

有 healthcheck 服务都返回“检查失败”时，Rancher 才会认为容器不健康。这避免了因网络抖动等原因造成检查结果不准确。

The screenshot shows the '健康检查' (Health Check) tab in the Rancher UI. The 'Type' is set to 'TCP连接' (TCP connection). The '端口*' (Port) is set to '例如:80'. The '初始化超时' (Init timeout) is 60000 ms, and '重新初始化超时' (Restart init timeout) is 60000 ms. The '检查间隔' (Check interval) is 2000 ms, and '检查超时' (Check timeout) is 2000 ms. 'Healthy After' is 2 successful checks, and 'Unhealthy After' is 3 failed checks. Under '不健康时' (When unhealthy), the '重新创建' (Restart) option is selected.

配置项	值	单位/说明
Type	TCP连接	
端口*	例如:80	
初始化超时	60000	毫秒
重新初始化超时	60000	毫秒
检查间隔	2000	毫秒
检查超时	2000	毫秒
Healthy After	2	次成功
Unhealthy After	3	次失败
不健康时	重新创建	

图 5-28 配置 TCP 方式的服务健康检查

The screenshot shows the '健康检查' (Health Check) tab in the Rancher UI. The 'Type' is set to 'HTTP响应2xx/3xx'. The '端口*' (Port) is set to '例如:80'. The 'HTTP请求*' (HTTP request) is set to 'GET', with a placeholder for the path '请求路径, 例如:/healthcheck'. The 'HTTP版本' (HTTP version) is set to 'HTTP/1.0'. The '初始化超时' (Init timeout) is 60000 ms, and '重新初始化超时' (Restart init timeout) is 60000 ms. The '检查间隔' (Check interval) is 2000 ms, and '检查超时' (Check timeout) is 2000 ms. 'Healthy After' is 2 successful checks, and 'Unhealthy After' is 3 failed checks. Under '不健康时' (When unhealthy), the '重新创建' (Restart) option is selected.

配置项	值	单位/说明
Type	HTTP响应2xx/3xx	
端口*	例如:80	
HTTP请求*	GET	请求路径, 例如:/healthcheck
HTTP版本	HTTP/1.0	
初始化超时	60000	毫秒
重新初始化超时	60000	毫秒
检查间隔	2000	毫秒
检查超时	2000	毫秒
Healthy After	2	次成功
Unhealthy After	3	次失败
不健康时	重新创建	

图 5-29 配置 HTTP 方式的服务健康检查

5.4.4 Rancher 的元数据服务

Rancher 的集群元数据信息是通过运行在每个 Agent 节点 `network-services` 应用栈中的 `metadata` 服务收集和管理的，通过 Rancher 的元数据服务，用户可以在任何由 Rancher 托管的容器内方便地获得当前容器的运行上下文信息，例如容器所属服务、应用栈和主机以及整个集群中所有服务和应用栈的清单。

另外值得一提的是，通过 Rancher 界面进入特定容器的详情页面后，可以通过其菜单中的“执行命令行”打开一个在容器内的 Web Console，从而方便地在容器内执行命令，如图 5-30 所示。

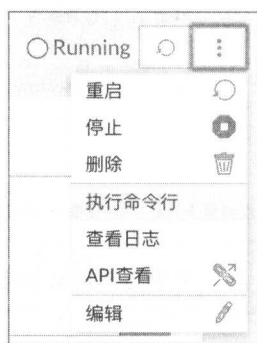


图 5-30 在界面上进入容器命令行

在容器中通过“`metadata.network-services`”地址可以访问 Rancher 的元数据服务。需要注意的是，在 Rancher 文档中使用的地址是“`rancher-metadata`”，但从实际测试结果来看，这个地址并不可用。例如以下请求将返回当前容器的名称。

```
$ curl http://metadata.network-services/2015-12-19/self/container/name
Demo-Demo-1
```

Rancher 元数据 API 的通用规则是“`http://metadata.network-services/<version>/<path>`”，其中 `<version>` 部分路径的规则如下所示。

- `<version>` 部分是一个用时间表示的 API 版本号，当前最新的版本是 `v2`，它在路径中的标识应该写为“`2015-12-19`”，也可以用“`latest`”表示始终使用最新版本（笔者个人不太赞同这种用发布时间来表示 API 版本的做法，但 Rancher 的确是这么实现的）。
- `<path>` 部分表示所需要查询的数据内容，可用的根路径及其描述如表 5-1 所示。

表 5-1 Rancher 的元数据查询路径及其描述

元 数 据	路 径	描 述
当前容器	self/container	当前执行元数据查询命令的容器自身的信息
容器所属的服务	self/service	当前执行元数据查询命令的容器所属服务的信息
容器所属的应用栈	self/stack	当前执行元数据查询命令的容器所属应用栈的信息
容器所运行的主机	self/host	当前执行元数据查询命令的容器所运行于主机的信息
集群中的容器	Containers	当前集群中所有容器的元数据信息
集群中的服务	Services	当前集群中所有服务的元数据信息
集群中的应用栈	stacks	当前集群中所有应用栈的元数据信息

这些元数据采用树形结构组织，访问其中的任意一个根都会返回该路径下可用的子级路径列表，例如“/self/container”路径下面包含了几十种具体属性，如下所示。

```
$ curl http://metadata.network-services/2015-12-19/self/container
create_index
dns/
dns_search/
environment_uuid
external_id
... ..
```

也可以指定使用 Json 格式获取元数据，这种方式能够一次性获得特定路径下的所有属性，如下所示。

```
$ curl --header 'Accept: application/json' http://metadata.network-services/2015-12-19/self/container
{
  "create_index": 6,
  "dns": [
    "169.254.169.250"
  ],
  "dns_search": [
    "demo.rancher.internal",
    "demo.demo.rancher.internal",
    "rancher.internal"
  ],
  ... ..
}
```


5.4.5 Rancher 的 DNS 服务

在第 5.3.3 小节关于服务别名的部分里已经简单地介绍过，通过 Rancher 中的 DNS 可以让一个服务通过名称直接访问同应用栈下的其他服务，其实这只是使用 Rancher DNS 服务进行域名解析访问的一种特例。

Rancher 服务名称映射功能是通过 Rancher DNS 服务实现的，它提供的功能类似第 4 章中介绍的 Mesos-DNS，就是将服务的名称与 IP 地址自动绑定起来。在主机的界面上将鼠标移到任意一个节点的“network-service”应用栈的“metadata”服务下的从容器上，就会看到这个从容器叫作“metadata-dns-1”，如图 5-31 所示。

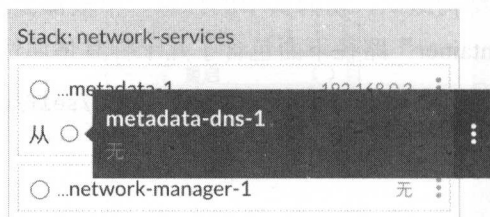


图 5-31 Rancher DNS 服务

在任何在 Rancher 中创建的服务都会自动获得一个集群内的唯一域名，使用“<服务名称>.<应用栈名称>.<Rancher 环境后缀>”表示，在同一个环境里的容器直接可以通过“<服务名称>.<应用栈名称>”的简化域名相互访问，而在同一个应用栈里的容器则可以通过更简单的“<服务名称>”直接找到对方。这实际上是通过每个容器里的“/etc/resolv.conf”文件配置实现的。

以 Demo 应用栈的 Demo 服务中的任意一个容器为例，它的“/etc/resolv.conf”文件第一行包含了一条域名搜索规则，如下所示。

```
$ cat /etc/resolv.conf
search demo.rancher.internal demo.demo.rancher.internal rancher.internal
nameserver 169.254.169.250
options timeout:1 attempts:1 rotate
```

根据这个规则，当用户以“DemoMaster”进行地址访问的时候，就会自动匹配到“DemoMaster.demo.rancher.internal”这条完整的域名记录，同样地，在上个小节中使用的元数据服务地址“metadata.network-services”的完整域名其实为“metadata.network-services.rancher.internal”。

5.4.6 使用私有镜像仓库

在部署服务时可能需要使用来自私有仓库的镜像，这些仓库可能需要登录验证。在第 3.3.5 小节中介绍过 Kubernetes 通过 Secret 对象来允许集群内的主机在下载这类镜像时自动进行身份验证，而 Rancher 则专门提供了一个模块用于管理用户的私有仓库资源。

单击 Web UI 的“基础架构”菜单的“镜像库”项目，进入镜像仓库管理页面。然后单击“添加镜像库”按钮。Rancher 默认内置了 DockerHub 和 Quay.io 的镜像仓库地址，也可以选择“Custom”来使用企业内部自己搭建的镜像仓库，如图 5-32 所示。



图 5-32 Rancher 支持的镜像仓库类型

在添加仓库页面里填写仓库地址、用户的账号和密码信息后单击“创建”即可。需要注意的是，在每个环境中，每个镜像仓库的地址对应的访问账号只能有一个，如果添加了多个，Rancher 将使用最新添加的那个。

此外，如果添加的仓库使用的是 HTTP 地址而不是 HTTPS，用户需要在每个主机上手动添加 Docker 后台服务的 `--insecure-registry` 启动参数，如下所示。

```
# 编辑 Docker 配置文件
$ sudo vi /etc/default/docker
# 在文件结束处添加下面这行
$ DOCKER_OPTS="$DOCKER_OPTS --insecure-registry=${DOMAIN}:${PORT}"
# 重启 Docker 服务
$ sudo service docker restart
```

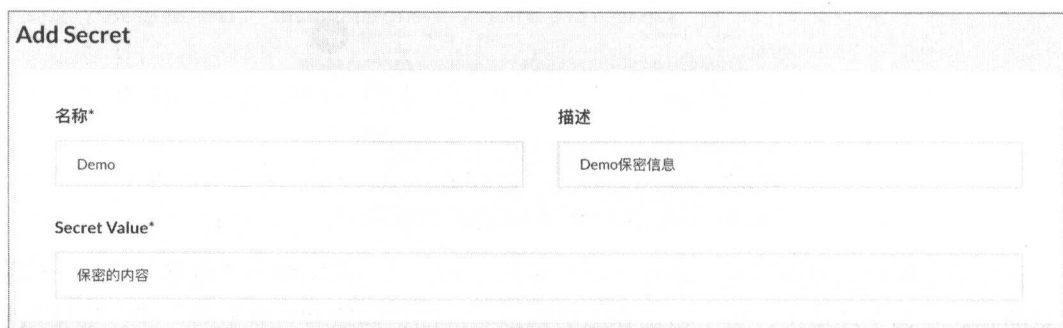
如果仓库使用了 HTTPS 地址，但用的是自签名的 SSL 证书，用户也需要在每个主机上手动添加相应的根证书文件，如下所示。

```
# 复制根证书文件到 Docker 证书目录下仓库域名对应的子目录
$ sudo cp ca.crt /etc/docker/certs.d/${DOMAIN}/ca.crt
# 附加根证书到系统的受信任证书文件末尾
$ cat ca.crt | sudo tee -a /etc/ssl/certs/ca-certificates.crt
# 重启 Docker 服务
$ sudo service docker restart
```

5.4.7 Rancher 的 Secret 服务

Rancher 的 Secret 服务的功能与 Kubernetes 的 Secret 对象差不多，都是为了将私密的信息从服务的容器镜像里面解耦出来，在运行时动态地加载，可以理解成是一种密码管理的机制，比如正式产品环境的数据库密码、某些重要系统的登录信息等，使用 Secret 可以有效避免保密数据的扩散。

单击 Web UI 的“基础架构”菜单的“Secret”项目，进入 Secret 的管理页面。然后单击“Add Secret”按钮来新增一个 Secret，如图 5-33 所示。



Add Secret

名称*

描述

Secret Value*

图 5-33 创建 Secret

在创建服务时，可以在 Secrets 面板里进行引用，并定义一个加载到容器中的文件名字，如图 5-34 所示。这样容器启动后就会自动挂载一个路径是“/run/secrets/demo”的文件，文件的内容就是引用的 Secret 的原始值。



命令 卷 网络 安全/主机 Secrets 健康检查 标签 调度

Secrets (+) Add Secret

Secret*

As Name

Secrets will be available in /run/secrets/ inside the Container with the given filename. Customize file ownership & permissions

图 5-34 使用 Secret

Secret 的值默认使用 AES256 算法加密存储在 Rancher 的数据库中，原则上来说，除非被挂载到具体的容器里，它的内容即使被截获也不会泄密，因此只要控制能够挂载 Secret 的用户权限就可以了。但实际上，这种默认加密方式的密钥也是存放在数据库里的，因此

数据库一旦被破解，所有加密信息还是存有泄露风险的。为此 Rancher 支持将 Secret 数据保存到 Vault^①工具里，Vault 是 Hashicorp 公司推出的一种专用于密钥存储的开源工具，它提供了安全可靠的密钥存储策略和密钥轮换机制，关于这个工具的使用并不在本书计划的范围里，因此不再具体展开。同时，Vault 与 Rancher 的集成方法这里也相应省略，读者如有需要可以参考相应文档^②的描述。

5.4.8 在应用商店添加自定义应用

前文已经介绍了 Rancher 应用商店的使用。倘若仅仅使用官方提供的有限的应用，是无法最大化发挥应用商店价值的。实际上，Rancher 的应用商店是一个很好的复用已有的劳动成果以及将自己的作品提供给其他用户使用的途径。在企业里，合理运用应用商店功能，还能实现发布服务的即时体验效果。

Rancher 应用仓库本质上是特殊目录结构的 Git 仓库。它要求在仓库的根目录下有一个名为“templates”的子目录，在这个目录下的每一个子目录都是一个应用。每个应用目录里都包括一个带“catalogIcon-”前缀的图片文件、一个“config.yml”文件和一系列以数字版本编号的目录，这些版本号标记的目录里，都包含“docker-compose.yml”和“rancher-compose.yml”两个文件。此外，推荐为每个应用添加一个介绍应用的“README.md”文件（非必须），最终形成以下结构。

```
templates
|--app-name
|  |-- 0
|  |  |-- docker-compose.yml
|  |  |-- rancher-compose.yml
|  |-- 1
|  |  |-- docker-compose.yml
|  |  |-- rancher-compose.yml
|  |-- catalogIcon-app-name.svg
|  |-- config.yml
|  |-- README.md
|--another-app-name
|  |-- ...
...

```

其中带“catalogIcon-”前缀的图片文件（可以是 svg、jpg、png 等常用格式）是应用

① <https://www.vaultproject.io/>

② <https://docs.rancher.com/rancher/v1.5/en/cattle/secrets/>

在 Rancher 应用商店显示的图标，如图 5-22 中每个应用的配图。“config.yml”是应用基本属性的配置文件，以 ZooKeeper 应用为例，如下所示。

```
$ cat zookeeper/config.yml
name: Apache Zookeeper
description: |
  (Experimental) Zookeeper cluster
version: 3.4.9-rancher1
category: Clustering
maintainer: "Raul Sanchez <rawmind@gmail.com>"
projectURL: https://github.com/rawmind0/alpine-zk
license:
```

每个属性的意义如下所示。

- **name**: 应用在界面上显示的名称。
- **description**: 应用在界面上显示的详细描述。
- **version**: 默认使用的版本，需要和“rancher-compose.yml”内的版本一致。
- **category**: 辅助搜索的元数据，表示应用的类别。
- **maintainer**: 项目维护者的信息。
- **projectURL**: 项目的网址。
- **license**: 项目采用的协议。

项目定义的子目录是以数字命名的，从数字 0 开始递增，其中的“docker-compose.yml”和“rancher-compose.yml”文件与 Rancher 服务编排所提到的应用描述文件基本相同，但在“rancher-compose.yml”文件中，除了普通的“services”，还需要一个“catalog”区域，包含对当前这个版本的补充描述信息，如下所示。

```
version: '2'
catalog:
  name: Zookeeper
  version: 3.4.9-rancher1
  description: |
    (Experimental) Apache Zookeeper cluster.
  minimum_rancher_version: v0.59.0
  maintainer: "Raul Sanchez <rawmind@gmail.com>"
  questions:
    - variable: "zk_scale"
      description: "Number of zk nodes"
      label: "Zk Nodes:"
      required: true
      default: 3
      type: "int"
    - ...
services:
```



```
zk:
```

```
...
```

其中“catalog”可用的属性有以下这些：

- **name**: 这个版本的应用名称。
- **version**: 版本标识，应该与“config.yml”文件里的版本采用相同的表示方法。
- **description**: 这个版本的应用描述。
- **minimum_rancher_version**: 支持的最低 Rancher 版本。
- **maximum_rancher_version**: 支持的最高 Rancher 版本。
- **upgrade_from**: 这个版本可以兼容从哪些版本进行升级。
- **questions**: 在创建应用时，允许用户修改的参数列表，包括以下子属性。
 - **variable**: 这个参数被引用时使用的名字。
 - **label**: 这个参数显示在界面上的提示信息。
 - **description**: 这个参数显示在界面上的描述信息。
 - **required**: 是否为必填参数。
 - **default**: 参数的默认值（如果是非必填参数，通常需要设置默认值）。
 - **type**: 参数的类型。

questions 这个属性相对比较复杂，但它是 Rancher 应用商店中重要的特性，使得用户在创建应用时能够自定义修改一些必要的配置，例如创建的副本数、连接的外部服务地址和账号等，这些信息是在定义通用应用模板时无法确定的。参数的类型可以是 **string**、**int**、**boolean**、**password**、**service**、**enum** 中的一种，其中 **password** 表示用户在界面输入参数内容时会显示为星号，**service** 表示此处只能选择一个在同应用内的服务名称，**enum** 则表示用户只能在给定的几个可选项中挑选一个。

在“questions”中定义的参数可以在“docker-compose.yml”和“rancher-compose.yml”文件的其他位置使用“\${参数名}”方式引用，例如这个例子的 **zk_scale** 参数在文件中引用时应该写成 **\${zk_scale}**。

单纯从描述来看，这样的仓库结构似乎略显复杂，不妨实际动手尝试一下，会更加直观。另外，网络上的有些社区 Rancher 应用仓库使用的是较早（v1 版本）的“docker-compose.yml”和“rancher-compose.yml”文件写法，与上述介绍的格式略有差异。Rancher 的官方应用仓库^①可以用来作为学习 Rancher 应用仓库结构和内容的参考范本。

准备完成自定义的应用仓库后，还要将其添加到 Rancher 里。单击“系统管理”下拉菜单中的“系统设置”子项，在系统设置页面中有一个“应用商店”的配置区域，在这里

① <https://github.com/rancher/rancher-catalog>

可以添加自定义仓库到 Rancher，如果需要的话，也可以在这里去掉内置的官方应用仓库，如图 5-35 所示。

应用商店

应用商店包含应用的rancher-compose模板，用户能够在回答设置问题后简单快速的进行应用部署。

Rancher Certified Library

☒启用 ☐禁用

Templates required for core Rancher features such as Kubernetes/Mesos/Swarm orchestration support. Maintained and supported by Rancher Labs.

社区贡献

☒启用 ☐禁用

由社区成员创建并维护的模板 未经过Rancher Labs认证

更多

此处可以添加自定义的应用商店。每个应用商店必须有唯一的名称和一个支持 git clone 操作的URL。 (更多信息请参考此文档)

+ 添加应用商店

Name

Demo

URL

https://github.com/...

Branch

master

保存

图 5-35 添加应用仓库

单击“保存”后，在“应用商店”菜单里就能看到新增加的仓库了，如图 5-36 所示。

图 5-36 应用商店列表中新增的仓库

5.5 Rancher 的命令行工具

5.5.1 配置 Rancher 命令行工具

虽然 Rancher 的 Web UI 功能十分丰富，但对于更习惯命令行操作的系统管理员，以及

268

在需要使用自动化运维的方式管理 Rancher 集群时，所有操作都需要打开浏览器并不是最佳的选择。

因此 Rancher 同样提供了一个简捷易用的命令行工具：`rancher`。单击界面右下角的“下载 CLI”按钮会展开下载菜单，可以直接单击相应操作系统的图标下载到本地，或者拷贝它的下载地址，然后在服务器下载它，如图 5-37 所示。

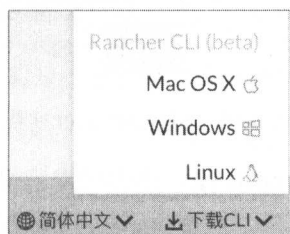


图 5-37 下载 Rancher 命令行工具

将下载后的压缩包解开，里面只有一个可执行文件，将这个可执行文件拷贝到系统 PATH 变量的目录里，如下所示。

```
$ wget https://releases.rancher.com/cli/v0.5.1/rancher-linux-amd64-v0.5.1.tar.gz
$ tar xzf rancher-linux-amd64-v0.5.1.tar.gz
$ sudo mv rancher-v0.5.1/rancher /usr/local/bin
```

在使用命令行工具前，需要告诉它应该连接哪个 Rancher Server，并让 Rancher Server 信任来自客户端的请求。因此需要配置 `rancher` 命令，给它一个身份授权的密钥。

在界面上单击“API”菜单，单击“Key”子项进入 API 密钥管理页面。用户可以在这里创建与当前登录账号相同权限的 API 密钥，如图 5-38 所示。

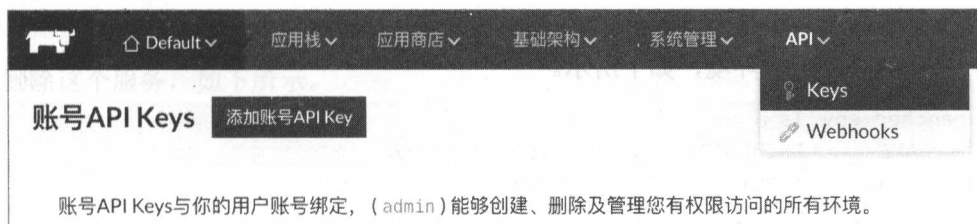


图 5-38 Rancher 的 API 密钥管理页面

单击“添加账号 API Key”按钮，在弹出的对话框输入密钥的名称和描述，单击“创建”。然后 Rancher 会展示一对密钥新生成的密钥内容，此时务必将这两个 Key 的值保存下来，一旦丢失是无法找回的，如图 5-39 所示。

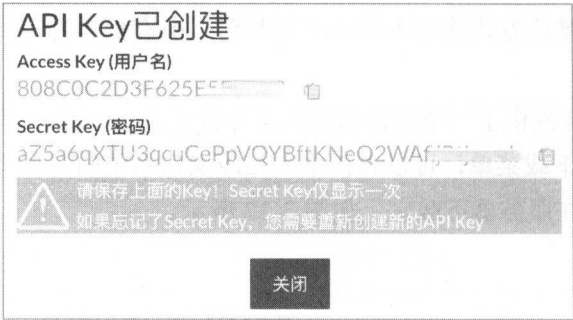


图 5-39 创建新的 API 密钥

回到控制台，执行 `rancher config`，然后按提示依次输入 Rancher Server 的地址、Access Key 和 Secret Key，如下所示。这样 Rancher 的命令行工具就配置好了。

```
$ rancher config
URL []: http://13.124.47.69:8080
Access Key []: 808C0C2D3F625E.....
Secret Key []: aZ5a6qXTU3qcuCePpVQYBftKNeQ2Waf.....
INFO[0116] Saving config to /home/ubuntu/.rancher/cli.json
```

Rancher 还曾有一个叫作“`rancher-compose`”命令行工具，它的功能已经被新的 `rancher up` 命令替代了，因此本书不再介绍与这个工具相关的内容。

5.5.2 命令工具的基本使用

Rancher 命令行工具虽然体积小巧，但其提供的功能完全不亚于 Web UI，熟练地使用这些命令能够有效提高集群管理的效率，下面举一些例子。

列出集群中的所有环境，如下所示。

```
$ rancher env ls
ID NAME ORCHESTRATION STATE CREATED
1a5 Default cattle active ...
```

列出集群中的所有主机，如下所示。

```
$ rancher host ls
ID HOSTNAME STATE CONTAINERS IP LABELS DETAIL
1h1 agent-1 active 10 13.124.44.61
1h2 agent-2 active 17 13.124.16.77
```

列出集群中的所有用户服务，如下所示。

```
$ rancher ps
ID   TYPE    NAME          IMAGE          STATE    SCALE  SYSTEM ...
1s6  service Demo/Demo     nginx:...     healthy   1/1    false  ...
```

列出集群中所有 Rancher 管理的用户容器，如下所示。

```
$ rancher ps -c
ID   NAME          IMAGE          STATE    HOST ...
1i10 rancher-agent rancher/agent:v1.2.1 running 1h2 ...
1i12 rancher-agent rancher/agent:v1.2.1 running 1h1 ...
1i20 Demo-Demo-1 nginx:1.11-alpine running 1h2 ...
```

除了查看集群的各类信息，Rancher 命令也可以对这些资源进行添加、删除和修改。例如在集群的指定环境中创建一个新的应用栈，如下所示。

```
$ rancher --env 1a5 stack create NewStack --empty
1st6
```

然后在这个应用栈里创建一个新的服务，如下所示。

```
$ rancher run --name NewStack/NewSrv --scale 2 nginx:1.11-alpine
1s7
```

再次查看服务列表，如下所示。

```
$ rancher ps
ID   TYPE    NAME          IMAGE          STATE    SCALE  ...
1s6  service Demo/Demo     nginx:...     healthy   1/1    ...
1s7  service NewStack/NewSrv nginx:...     healthy   2/2    ...
```

停止刚刚创建的服务，如下所示。

```
$ rancher stop 1s7
1s7
```

删除这个服务，如下所示。

```
$ rancher rm 1s7
1s7
```

这里单独介绍一下 **rancher docker** 命令，它可以远程在集群中任意主机上执行原生的 Docker 命令，在集群节点比较多时十分有用，例如查看指定主机中运行的容器，如下所示。

```
$ rancher --host=1h1 docker ps
CONTAINER ID  IMAGE          COMMAND          ...
1dae717ed3db rancher/net:v0.9.4 "/rancher-entrypoi..." ...
... ..
```


表 5-2 展示了 Rancher 命令行中的主要命令及其描述。其中删掉了一些功能重复的子命令，如 `rancher environment` 与 `rancher env` 命令等效，在表格中只保留比较常用的 `rancher env`。

表 5-2 Rancher 命令行工具的子命令列表

名 称	描 述
catalog	处理与应用商店相关的操作
config	配置命令行工具的目标服务器和身份验证密钥
docker	在集群的任意主机上执行 Docker 命令
env	处理与环境相关的操作
events	展示指定资源的事件日志
exec	在任意指定容器中执行指定的命令
export	将指定应用栈导出成 YAML 描述文件
hosts	处理与主机相关的操作
logs	获取任意指定容器的输出日志
ps	获取所有服务和容器的列表
restart	重启指定的服务或容器
rm	删除指定资源（包括主机、应用栈、服务、容器或存储卷）
run	根据指定的参数信息创建服务
scale	增加或减少指定服务的容器副本数目
ssh	快速 SSH 登录到集群中的指定主机
stack	处理与应用栈相关的操作
start	启动指定的服务或容器
stop	停止指定的服务或容器
up	通过应用编排文件快速创建或升级应用栈
volume	处理与存储卷相关的操作
inspect	查看资源的详细信息（包括环境、主机、应用栈、服务、容器或存储卷）
wait	等待指定资源的当前操作完整（包括主机、应用栈、服务、容器等）
help	显示可用命令的列表和帮助信息

此外，Rancher 命令行有一些全局的参数，在使用时需注意，如下所示。

- `--env`: 指定命令执行的环境，如未指定，则默认为 `Default` 环境。
- `--host`: 指定命令执行的主机，目前仅对 `rancher docker` 命令有用。
- `--wait`: 等指定资源就绪再执行这条命令。
- `--wait-state`: 指定“就绪”的条件，如 `active`、`healthy` 等。
- `--wait-timeout`: 指定等待资源就绪的超时时长（以 `s` 为单位），默认是 `600s`。

5.5.3 通过命令行进行服务编排

在第 5.3.5 小节里已经介绍过在 Rancher 中描述服务编排的方式。在 Rancher 的命令行里，用于操作服务编排的命令是 `rancher up`，它使用与 Rancher Web UI 相同的输入文件：“`docker-compose.yml`”和“`rancher-compose.yml`”。

为了便于演示，创建一个极简的应用描述文件，如下所示。

```
$ cat <<EOF >docker-compose.yml
version: '2'
services:
  NginxSrv:
    image: nginx:1.11-alpine
EOF

$ cat <<EOF >rancher-compose.yml
version: '2'
services:
  NginxSrv:
    scale: 2
EOF
```

上述文件描述了一个只有一个服务的应用栈，这个服务使用 `nginx:1.11-alpine` 作为镜像，创建两个容器副本，其他配置全部使用默认值。

在同一个目录里执行 `rancher up` 命令，使用 `--stack` 或 `-s` 参数指定要创建的应用名称，`-d` 参数表示在后台启动完成后就立即返回命令行，如下所示。

```
$ rancher up -s NginxStack -d
```

`rancher up` 命令默认使用当前目录的 `docker-compose.yml` 和 `rancher-compose.yml` 文件作为输入，如果这两个文件存放在其他的位置，可以通过 `--file` 和 `--rancher-file` 参数明确指定出来。

稍等片刻后，一个名称是“`NginxStack`”的应用栈就创建完成了，同时应用栈中已经运行了编排描述文件中指定的 `NginxSrv` 服务。

5.5.4 通过命令行进行服务升级

在命令行中对应用栈和服务进行升级的命令也是 `rancher up`，只需要添加一个额外的 `--upgrade` 或 `-u` 参数。此时通过 `--stack` 或 `-s` 参数指定的应用栈必须已经存在，Rancher

会将该应用栈当前的配置与新的编排描述文件进行对比，并更新其中不一致的部分。

首先修改服务编排文件，例如替换“docker-compose.yml”文件中的镜像版本，然后执行以下命令对刚刚创建的应用栈进行升级。

```
$ rancher up -s NginxStack -d -u
```

与通过界面执行的升级操作一样，当新版本的服务容器就绪后，Rancher 不会立即删除旧版本的容器，而是等待用户进行确认。

确认的方法是向目标应用栈使用 `--confirm-upgrade` 或 `-c` 参数的 `rancher up` 命令，升级确认后，Rancher 将删除旧版本容器，如下所示。

```
$ rancher up -s NginxStack -d -c
```

如果此时用户希望取消这次升级，可以对升级操作进行回滚。方法是向目标应用栈使用 `--rollback` 或 `-r` 参数的 `rancher up` 命令，如下所示，执行回滚后，应用栈将恢复到升级前的状态。

```
$ rancher up -s NginxStack -d -r
```

使用 `rancher up` 命令升级应用栈时，Rancher 同样是采用滚动升级方式分步进行的，可以通过参数修改滚动升级时每批更新的容器数量以及升级批次间隔的时间（以 `ms` 为单位），如下所示。

```
$ rancher up -s NginxStack -d -u -c --batch-size 1 --interval 1000
```

5.6 使用 Rancher 安装 Kubernetes

5.6.1 Rancher 的环境管理

作为一种容器任务调度平台，Rancher 在设计之初预留下了一个对未来扩展相当有用的概念：环境。在一个刚刚创建好的集群里，Rancher 自动为用户创建好了名称是“Default”的环境，它使用默认的 Cattle 引擎作为任务调度和管理的工具。在其页面左上角的第一个菜单里可以看到与环境相关的子项，如图 5-40 所示。

在早期的 Rancher 里，环境的作用仅仅是作为逻辑租户隔离的手段，不同的环境代表着不同的机房、不同用途或者不同用户的资源。从 1.0 版本开始，Rancher 逐渐增加了

Kubernetes、Swarm（现在已经替换为 SwarmKit）、Mesos 等第三方编排引擎的支持，为了使这些编排引擎运行时不至于相互“打架”，对不同引擎的集群进行物理主机的隔离变得尤为重要，此时 Rancher 中每个环境独占资源的优势就充分体现出来了。

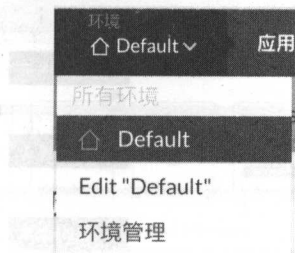


图 5-40 Rancher 的环境菜单

Rancher 能够“替换”调度引擎，这和 Mesos 的两层调度思想有什么不同呢？从原理上来看，Rancher 的资源层虽然可以事后重新分配，但并不是自动动态调配的，因此不能算作“两层调度”，而是“一层分配加一层调度”，可看作是 Mesos 模型的一种简化实现版，如图 5-41 所示。

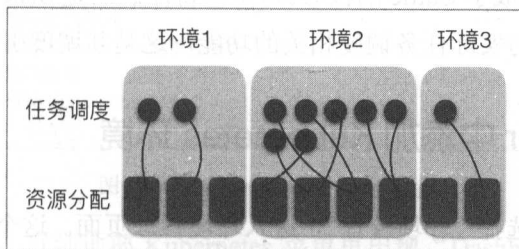


图 5-41 Rancher 的资源和任务管理

简化了是不是说明 Rancher 的模型不够好呢？事实上恰恰相反。

Mesos 社区曾经多次尝试将 Kubernetes 和 Swarm 作为一种任务调度器加入它的统一模型里，但最终都以太复杂和不稳定而不了了之。Rancher 仅仅花了不到 1 年的时间，就把各种主流调度引擎的集成全部搞定。由此看来，Rancher 在设计取舍上也可以算有一种大智慧。

在每个环境里，Rancher 都会在集群的主机中预装一些基础架构服务。以默认的 Cattle 引擎为例，Rancher 默认创建了如图 5-42 所示的 4 个应用栈。

- heathcheck: 提供用户服务健康检查的功能。
- ipsec: 提供能够跨节点通信的 Overlay 容器网络。

- network-services: 提供负载均衡、DNS 等网络功能。
- scheduler: 提供用户任务调度的功能。



Infrastructure Stacks		Add from Catalog	排序:	状态	名称
+	healthcheck	已经是最新版本	1 服务	1 容器	
+	ipsec	已经是最新版本	1 服务	3 容器	
+	network-services	已经是最新版本	2 服务	3 容器	
+	scheduler	已经是最新版本	1 服务	1 容器	

图 5-42 Cattle 环境的基础架构服务

还有一些服务（例如存储相关的服务和扩展的网络服务）是需要在应用市场以插件的方式安装的, 这些服务构成了 Cattle 的核心。这里值得注意的是 Scheduler 这个服务, Rancher 以单独服务的形式提供与实际任务调度相关的功能, 这是其调度引擎能够替换的关键。

5.6.2 在 Rancher 中添加 Kubernetes 环境

在环境下拉菜单中选择“环境管理”，进入环境管理页面。这个页面包含两个部分，即环境的管理和环境模板的管理。在 Rancher 中，所谓的环境类型，实际上就是一些由基础架构服务组成的集群配置，而这些配置就是通过环境模板的方式管理的。

Rancher 里内置了 5 种基本的模板，单击页面的“添加环境”按钮进入添加环境页面，如图 5-43 所示。这 5 种模板涵盖了绝大多数用户所需的 5 种典型集群环境：完全的 Cattle 环境、使用 Kubernetes 作为调度器的环境、使用 Mesos 作为调度器的环境、使用 SwarmKit 作为调度器的环境以及使用 Windows 版 Docker 组成的环境。因此本书不对环境模板管理的部分进行详细介绍，如果读者有对环境进行深度定制的需求，可以参考相应文档内容。

在添加环境页面里，选择 Kubernetes 类型的环境，指定一个名称，单击“创建”按钮。然后从环境菜单中切换到新建的 k8s 环境，并向这个环境中添加一些主机。此时 Rancher 主界面上会出现 Kubernetes 集群正在创建的信息，如图 5-44 所示。

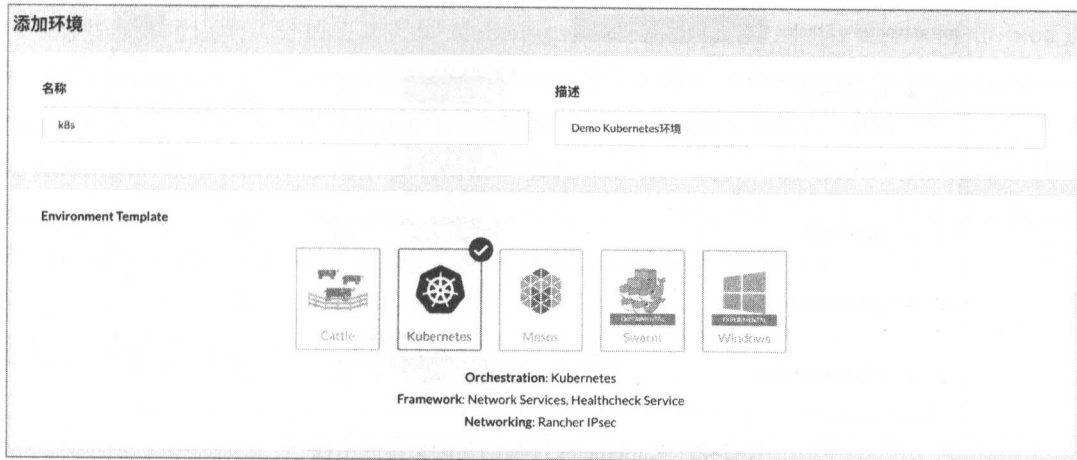


图 5-43 Rancher 的添加环境页面

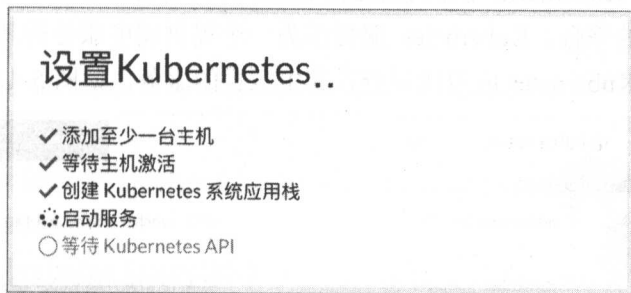


图 5-44 Kubernetes 初始化创建

等待大约十几分钟，直到顶部 Kubernetes 菜单里出现“Dashboard”和“CLI”两个子项，如图 5-45 所示。

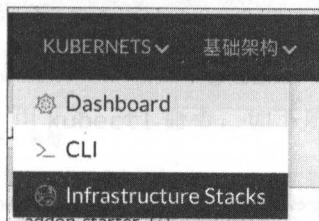


图 5-45 Kubernetes 菜单

查看集群此时的应用栈，会看到与 Cattle 类型环境十分相似的列表，其中增加了与 Kubernetes 相关的两个应用栈，而移除了在 Cattle 类型环境里的 Scheduler 应用栈，如图 5-46 所示。

Infrastructure Stacks		添加应用栈	Add from Catalog	排序:	状态	名称
+	healthcheck	已经是最新版本	1 服务	1 容器		
+	ipsec	已经是最新版本	1 服务	3 容器		
+	kubernetes	已经是最新版本	10 服务	12 容器		
+	kubernetes-ingress-lbs	添加服务	0 服务	0 容器		
+	network-services	已经是最新版本	2 服务	3 容器		

图 5-46 Kubernetes 环境的基础架构应用栈

换言之,这个Kubernetes环境仍然是在Cattle的核心设置基础上运行的,并由Rancher作为计算资源分配的平台。Kubernetes服务作为一个提供调度服务的基础架构应用栈被添加到集群里。单击Kubernetes应用栈后查看其中包含的服务,如图5-47所示。

应用栈:

kubernetes

添加服务

Active	addon-starter ①	镜像: rancher/k8s:v1.5.4-rancher1-3
Active	controller-manager ①	镜像: rancher/k8s:v1.5.4-rancher1-3
Active	etcd + 1 从容器 ①	镜像: rancher/etcd:v2.3.7-11
Active	kubectld ①	镜像: rancher/kubectld:v0.5.5
Active	kubelet ①	镜像: rancher/k8s:v1.5.4-rancher1-3
Active	kubernetes + 1 从容器 ①	镜像: rancher/k8s:v1.5.4-rancher1-3
Active	proxy ①	镜像: rancher/k8s:v1.5.4-rancher1-3
Active	rancher-ingress-controller ①	镜像: rancher/lb-service-rancher:v0.6.1
Active	rancher-kubernetes-agent ①	镜像: rancher/kubernetes-agent:v0.5.4
Active	scheduler ①	镜像: rancher/k8s:v1.5.4-rancher1-3

图 5-47 Kubernetes 应用栈中的服务

这些服务可以大致分成三类，即 Kubernetes 的原生服务、Rancher 适配服务和 Etcd 服务。其中 Kubernetes 的原生服务包括提供 Master 节点功能的 kube-apiserver、kube-scheduler、kube-controller-manager，提供 Node 节点功能的 kubelet 和 kube-proxy。Rancher 适配服务是让 Kubernetes 能够从 Rancher 中获取信息和资源、进行中间协调的服务，包括 kuberctrld、ingress-controller、kubernetes-agent。同时这些服务还会依赖于每个主机上部署的其他基础设施应用栈所提供的 Overlay 网络、DNS、Metadata 等服务等功能。

5.6.3 在 Rancher 中使用 Kubernetes

环境部署完成后，单击“Kubernetes”菜单中的“CLI”子项，进入 Web 命令行窗口，如图 5-48 所示。

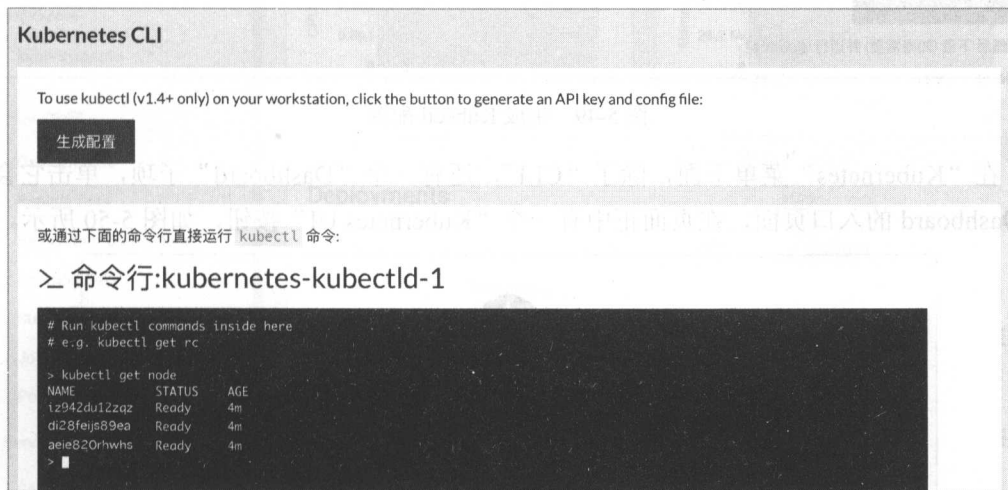


图 5-48 Kubernetes Web 命令行

在这个命令行中可以直接使用 `kubectl` 命令，如下所示。

```
$ kubectl get node
```

此外，也可以使用原始的 Kubectl 工具来操纵这个集群。Rancher 在创建 Kubernetes 集群时自动创建了一个授权用户，单击“生成配置”按钮，文本框中会显示连接这个 Kubernetes 环境的所需的配置，如图 5-49 所示，拷贝这些内容到客户端主机 `kubectl` 命令的默认配置文件位置（“`$HOME/.kube/config`”文件），然后就可以直接使用 `kubectl` 命令向普通的 Kubernetes 集群那样进行操作了。

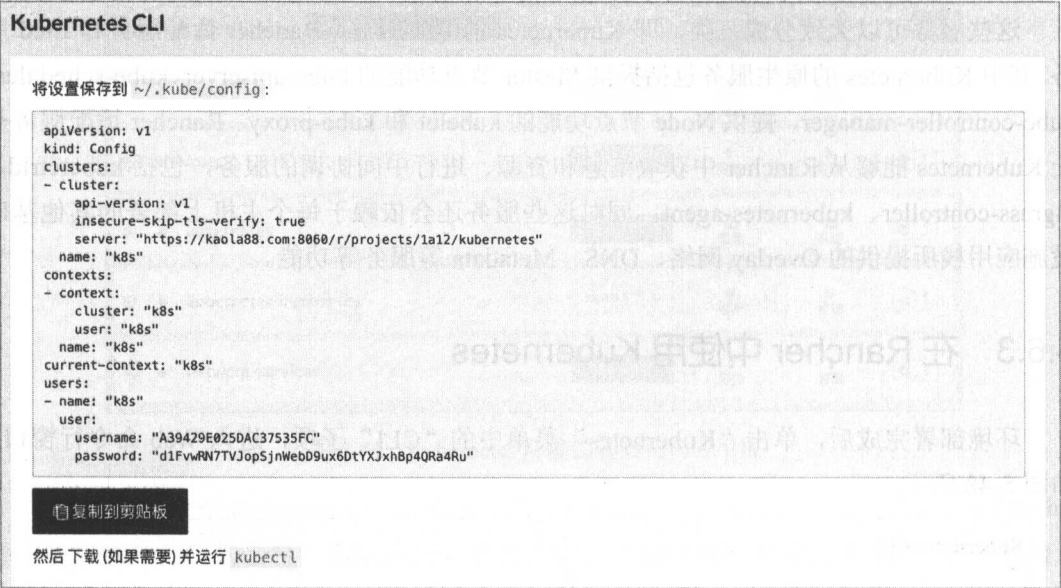


图 5-49 生成 Kubectl 配置

在“Kubernetes”菜单下面，除了“CLI”，还有一个“Dashboard”子项，单击它会打开 Dashboard 的入口页面，在页面正中有一个“Kubernetes UI”按钮，如图 5-50 所示。

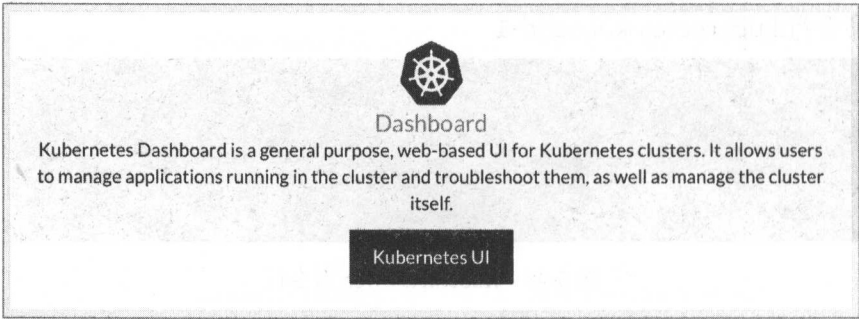


图 5-50 Kubernetes Dashboard 的入口页面

单击这个按钮就会跳转到 Kubernetes 的 Dashboard。若当前集群中还没有创建服务，可以从左侧的 Namespace 选择区切换到包含系统组件的 kube-system 空间，此时右侧会显示出在当前 Namespace 中的所有 Kubernetes 资源对象。注意右侧顶部的 CPU 和内存使用情况的报表，这个数据是通过各个节点 Kubectl 内置的 cAdvisor 服务发送到集群中的 Heapster 服务，然后上报到 Dashboard 的，如图 5-51 所示。

与 Dashboard 一样，Heapster 在 Kubernetes 中也是以独立插件的形式存在的。Rancher

在部署完 Kubernetes 集群后，顺便将常用的插件也都预装好了。为了更好地适配 Rancher 的运行环境，这些插件没有直接使用原生的部署描述文件，而是使用 Rancher 来维护在 GitHub 里 `kubernetes-package` 项目^①中的文件版本。在这个仓库中也可以看到除了 Dashboard 和 Heapster，Rancher 内置的插件还包括应用包管理工具 Helm，可以在 Web 命令行中输入例如“`helm version`”“`helm search`”等命令验证。

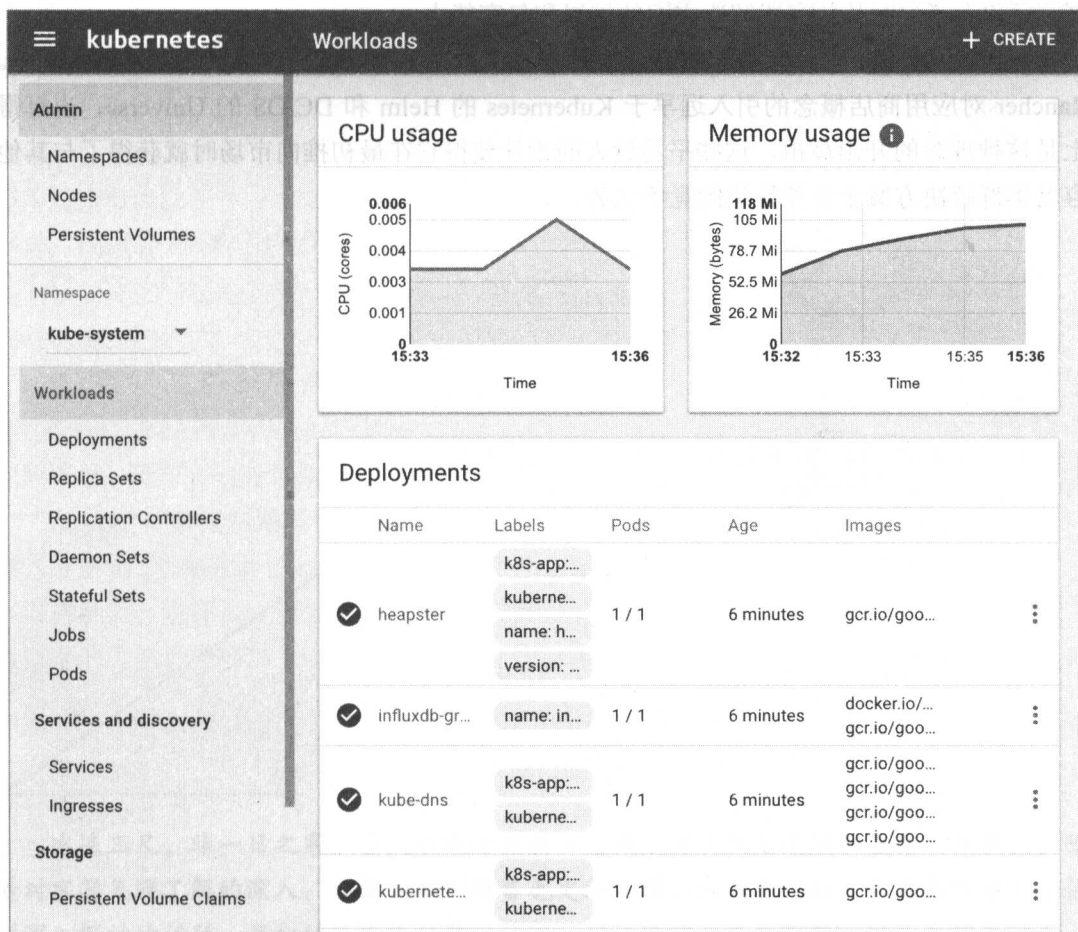


图 5-51 Rancher 部署的 Kubernetes Dashboard 界面

① <https://github.com/rancher/kubernetes-package>

5.7 本章小结

本章从 Rancher 项目的组成结构出发，详细介绍了这款颇具特立独行气质的容器集群方案的部署和操作方法，在最后的部分以通过 Rancher 部署并管理 Kubernetes 集群为例，展示了 Rancher 对其他容器解决方案的扩展和包容能力。

Rancher 提供了十分易用的 Web UI，用户通常在界面上就能完成集群管理的各种功能。Rancher 对应用商店概念的引入远早于 Kubernetes 的 Helm 和 DC/OS 的 Universe，也算得上是这种理念的开山鼻祖。这些平易近人的设计使得它在最初推向市场时就获得了与其他容器集群解决方案十分差异化的竞争优势。

第3部分 深入探索

冰冻三尺，非一日之寒。各式解决方案精心隐藏了许多底层烦琐和复杂的细节，但随着对容器集群了解的深入，所需的知识储备也更加广泛。网络和存储是容器集群中十分值得深入探讨的话题，类似的还有集群监控管理、日志管理、服务管理、镜像管理等方面。此外，在容器技术核心领域的边缘地带，还有更多极具潜力的交叉领域值得我们去探索。

- 容器集群的网络和存储
- 容器服务的基础设施
- 容器技术新风向

第 6 章 容器集群的网络和存储

在容器的大规模应用方面，有两个经常被提起的话题，它们就是容器的网络和存储。Docker 容器在最初设计时是没有为集群和分布式考虑的，以至于随着容器应用的普及，很多集群场景的问题纷纷暴露出来：跨节点网络无法互通、容器重启后漂移到其他节点导致存储数据丢失，以及容器被分配到的节点磁盘空间不足导致频繁崩溃等。

本章将讨论容器的网络和存储两方面的话题，并重点介绍“跨节点网络”和“跨节点存储”的实现。

6.1 容器网络

6.1.1 容器网络标准

作为集群的底层基础设施之一，网络的复杂性远远超出了 Docker 这类容器工具所能管控的范畴，但它也是企业级应用通往规模化无法绕过的一道坎。以 Docker 为例，在早期的 Docker 版本里，对于网络的支持仅限于“Bridge”“Host”和“None”这三种本地网络模式（实际上还有一种多个容器共享 Network Namespace 的 Container 模式，它是 Bridge 模式的一种特殊情况），若两个不同节点上部署的容器要通信，只能通过节点主机的 IP 地址和容器暴露到主机上的端口进行。

这种原始的容器通信方法在小规模应用容器的时候，只需提前设计好每个服务容器的运行节点和监听端口，是可以满足需要的。然而当容器规模逐渐增加，管理的成本很快成倍地增长，特别是自动扩缩容这样的场景出现时，容器运行节点和端口不变的前提就被完全打破了。大约在 2014 年以后，越来越多的用户开始使用动态的容器调度，对容器具有可跨节点路由 IP 地址的需求十分强烈。在这一阶段里，一些大量使用容器技术的企业各显神通，为社区贡献了许多早期的开源跨节点容器网络方案，这些方案大多是通过预先规划网

络地址结合辅助包转发实现的。这个时期的工具可以大致分为两类：一类能够自动规划网络地址，使用门槛较低，典型的例子如 Flannel 和 Weave；另一类则需要手动规划网络地址，提供高度定制化的用户灵活性，典型的例子如 OVS 和 Pipework。它们的实现原理比较相似，且都不依赖于容器工具（如 Docker）本身，仅仅在容器所使用的网桥上动手脚，从而将不同节点上的容器编排到一个虚拟的网络上，间接地实现了能够跨节点通信的容器 SDN 服务，在稍后的小节里会详细介绍它们的原理。

这种放任社区发展的阶段持续了接近 1 年。2015 年 3 月，Docker 收购了这些跨主机容器 SDN 网络解决方案的提供者之一：创业公司 SocketPlane。此前 SocketPlane 设计出了在 socket 层为 Docker 容器创建网络抽象层的产品，可以在一个逻辑网络内创建多个容器，这样的网络可以存在于一个单独的地址空间内，而且其实现不依赖于特定的物理网络控制器。随后，Docker 公司在博客里^①宣布了一个新项目：libnetwork。几个月后，Docker 1.7 版本发布，标志着 libnetwork 代码完全从原先的 lincontainer 仓库剥离，容器的网络接口在 libnetwork 中成为了独立可替换的插件模块，为之后的容器网络技术快速演进铺平了道路。

概括来说，libnetwork 所做的最核心的事情就是提出了第一个开放的标准网络模型的标准：Container Network Model（简称 CNM）。只要符合这个模型的网络接口就能被用于容器之间通信，通信的过程和细节可以完全由网络接口来实现。

Docker 的容器网络模型最初是由思科公司员工 Erik 提出的设想，比较有趣的是 Erik 本人并不是 Docker 和 libnetwork 代码的直接贡献者。最初 Erik 只是为了扩展 Docker 网络方面的能力，设计了一个 Docker 网桥的扩展原型，并将这个思路反馈给了 Docker 社区。然而他的大胆设想得到了 Docker 团队的认同，并在与 Docker 的其他合作伙伴广泛讨论之后，逐渐形成了 libnetwork 的雏形。

在这个网络模型中定义了几个术语：Sandbox、Endpoint 和 Network。它们分别是容器通信中“容器网络环境”“容器虚拟网卡”和“主机虚拟网卡 / 网桥”的抽象，如图 6-1 所示。

- **Sandbox**：对应一个容器中的网络环境，包括相应的网卡配置、路由表、DNS 配置等。CNM 很形象地将它表示为网络的“沙盒”，因为这样的网络环境是随着容器的创建而创建，又随着容器销毁而不复存在的。
- **Endpoint**：实际上就是一个容器中的虚拟网卡，在容器中会显示为 eth0、eth1，后面可以此类推。

^① <https://blog.docker.com/2015/04/docker-networking-takes-a-step-in-the-right-direction-2/>

- Network: 指的是一个能够相互通信的容器网络，加入了同一个网络的容器可以直接通过对方的名字相互连接。它的实体其实是主机上的虚拟网卡或网桥。

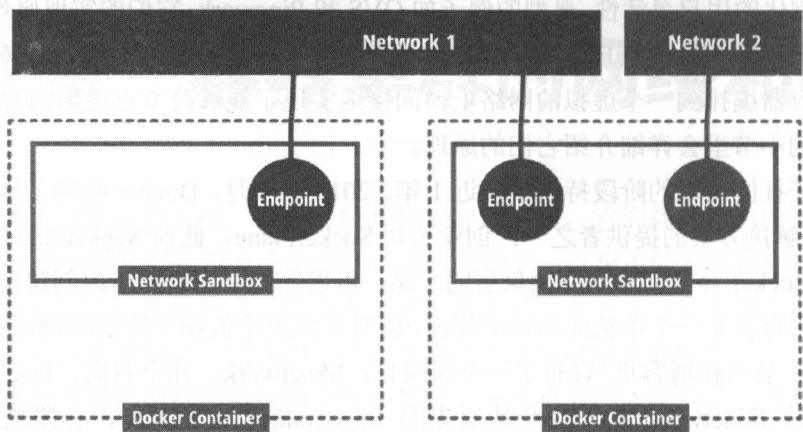


图 6-1 CNM 的网络模型示意

CNM 模型的抽象给 Docker 网络组件带来了十分平滑的过渡，除了文档中的三种经典“网络模式”被换成了“网络插件”，用户完全感觉不到使用起来有什么差异，如图 6-2 所示。

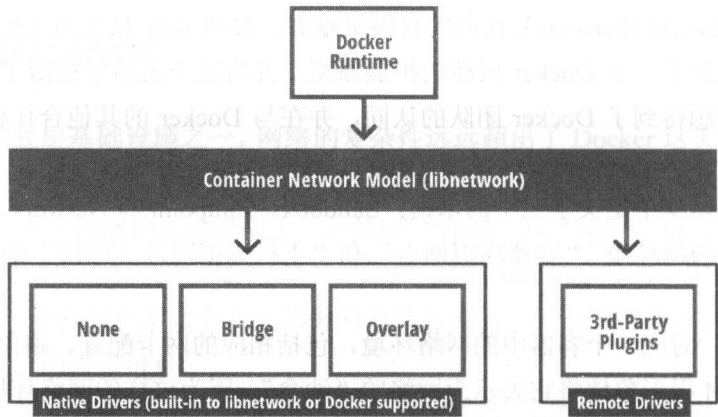


图 6-2 Docker 由过去的三种网络模式现在变成内置的三种网络插件

这个标准看似合理，但推出后不久社区里就出现了反对的声音，Kubernetes 平台甚至专门写了一篇博客^①来解释为什么它们决定不支持 CNM 模型。大多数 CNM 标准的反对者

① <http://blog.kubernetes.io/2016/01/why-Kubernetes-doesnt-use-libnetwork.html>

都是出于同样的一个原因，那就是这个标准太专制了。要知道 libnetwork 毕竟是 Docker 一家企业决策的，其中包含了许多并非基础容器网络必须的组件，因此 CNM 标准实际上是为 Docker 量身定做的，接纳 CNM 就意味着排除 Docker 以外的所有容器工具。

后来，这些不看好 CNM 的企业和开发者们转向了另一种轻量级应用容器网络的开放插件化标准：Container Networking Interface（简称 CNI）^①。CNI 标准主要是由 CoreOS 公司提出的 AppC 应用容器标准的一部分（Networking Proposal）发展而来。相比于 CNM 标准，CNI 本身并不针对 Docker 或是某种特别的容器，是一种普适的容器网络开放标准。CNI 的网络模型如图 6-3 所示，这个模型也比 CNM 更加简单，只有两个主要概念。

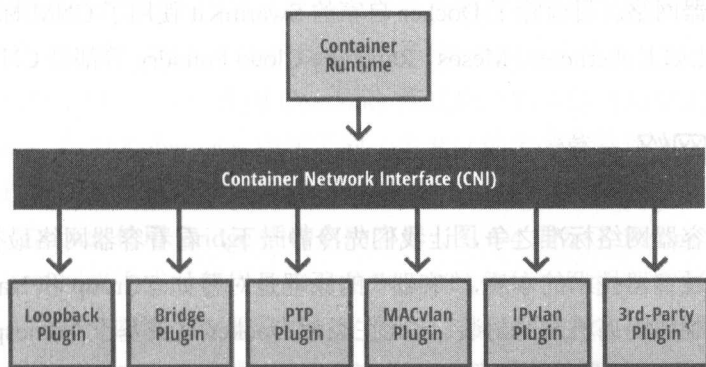


图 6-3 CNI 的网络模型示意图

- **Container**：指拥有独立 Network Namespace 的运行时单元，不论是 Rkt、Docker 还是 VM 原理创建的容器。拥有独立的 Network Namespace 是能够赋予独立网络地址和可进行路由的必要条件。
- **Network**：指可以相互联系的一组实体。这些实体拥有各自独立唯一的 IP 地址。实体可以是容器、物理机或者其他网络设备（比如网关、路由器等）。

从模型概念的角度上来看，CNI 中的 Container 概念与 CNM 的 Sandbox 概念基本一致，CNI 中的 Network 与 CNM 中的 Network 说的也是同一个东西（两者接口有差异）。只不过 CNM 模型里的 Endpoint 概念在 CNI 模型中被隐含在 Network 的操作中了。从模型实现的角度上来看，CNM 和 CNI 标准都提供了独立的扩展点，开发者可以通过插件机制为容器创建网络栈。两者都支持每个容器被加入到被不同插件所驱动多个网络之中，且每个网络有自己对应的插件和唯一的名称。不过在标准的接口方面，CNM 标准定义了十几个需要

^① <https://github.com/containernetworking/cni/blob/master/SPEC.md>

插件实现的 API 接口^①，且要求插件开发者实现一个提供网络 Socket 通信功能的后台服务。CNI 标准只有三个接口（添加网络、删除网络和查询版本）以及一个 Json 格式的配置文件，并不需要额外的后台服务，开发者可以通过任何编程语言编写可执行程序，甚至只需用 Shell 脚本就能完成，比如 Kubernetes 开发者提供了一个利用 `docker network` 命令实现的只有几十行 Shell 代码的 CNI 插件^②。

本质上说，CNI 和 CNM 的模型上并没有存在根本性冲突。虽然这种差异的存在确实为网络插件的开发者带来了一些麻烦，但目前看起来，有不少企业级的容器网络方案都同时适配了这两种标准，比如 Open Virtual Networking（即 OVN）、Calico、Weave 以及 VMware 的一些企业级容器网络。目前除了 Docker 自家的 SwarmKit 使用了 CNM 标准，其余的主流容器集群架构，比如 Kubernetes、Mesos、Rancher、Cloud Foundry 等都是 CNI 标准的支持者。

6.1.2 本地网络

说了半天的容器网络标准之争，让我们先冷静一下，看看容器网络最初的样子。

第 1 章介绍过容器技术的本质，“容器”的原理是对譬如 CGroup 和 Namespace、chroot 等内核级访问控制和隔离机制的封装。因此在类似 Docker 这类基于 Namespace 实现隔离的容器里，容器网络就是给容器中运行的服务赋予的独立 Network Namespace。这里有一个常见的误区，即认为容器的 Network Namespace 和主机的 Network Namespace 是父子层级关系。实际上，所有容器的 Namespace 和主机所在的默认 Network Namespace 在逻辑上是平等的，它们只是各自具有独立的 IP 地址，并可以分别关联主机的不同网络设备。

除了内核的 Network Namespace，容器网络还用到了 Linux 网络中的一些常见技术，比如 Network Bridge、Iptables、Veth Pair 等，其作用如下所示。

- **Virtual Bridge**: 一种虚拟的网络交换逻辑单元，功能相当于物理交换机，为连在其上的设备（容器）转发数据帧，常见的有 Docker 默认创建的 `docker0` 网桥。
- **Iptables**: 一种内置在内核中基于规则的网络防火墙和包转发服务，主要为容器提供与 NAT 以及容器网络安全相关的特性。
- **Veth Pair**: 由两个虚拟网卡 / 网桥组成的数据通道。例如在 Docker 中，用于通过一对 veth pair，一端在容器中作为 `eth0` 网卡，另一端链接到主机的 `docker0` 网桥，即可实现容器与主机的消息互通。

① <https://github.com/docker/libnetwork/blob/master/docs/remote.md>

② https://github.com/kubernetes/contrib/tree/master/cni-plugins/to_docker

通过这些技术, Docker 提供了三种内置的本地网络插件, 其他的容器项目 (如 Rkt 等) 对本地网络的支持也大致相同。这三种插件 (各自代表一种构建网络的模式) 分别如下所示。

- Host 网络插件: 通过这个插件启动的容器将和宿主机共享相同的 Network Namespace, 即直接使用宿主机的 IP 地址、端口、网络设备等资源。
- None 网络插件: 通过这个插件启动的容器将使用独立的 Network Namespace, 但并没有对其进行任何网络设置, 如分配 veth pair 和网桥连接、配置 IP 等。
- Bridge 网络插件: 通过这个插件启动的容器将使用独立的 Network Namespace, 同时 Docker 会自动为其创建与某个网桥 (默认网络中是 docker0) 连接的 veth pair, 并使用内置 IPAM 驱动为容器接口分配与网桥在相同子网段的私有 IP 地址。

除了这几个插件以外, 容器的本地网络还有一种经典的使用模式, 成为 Container 模式, 它不需要额外的网络插件来实现。使用 Container 模式启动的容器将和指定的其他容器共享 Network Namespace。例如 Kubernetes 中属于同一个 Pod 的容器就是通过这种模式使得多个容器能够直接通过 localhost 地址相互通信的。

默认情况下, Docker 使用 Bridge 插件启动容器, 图 6-4 展示了 Bridge 插件创建的容器的网络连接图。这种网络形式对使用者十分友好, 容器的每个网桥在会主机内部创建一个私有网络, 因此在同一个网络中容器可以相互通讯。Docker 容器在创建时可以指定一个需要连接的网络 (使用 `--network` 参数), 只有在同一网络内的容器可以通过 IP 地址通信, 外部及不同网络之间通信需通过映射到主机的端口进行 (使用 `--publish` 或 `-p` 参数)。这样可以限制不同容器之间的网络连接, 还可以设置访问规则来进一步阻止未授权的网络访问, 在一定程度上提升系统的安全性。

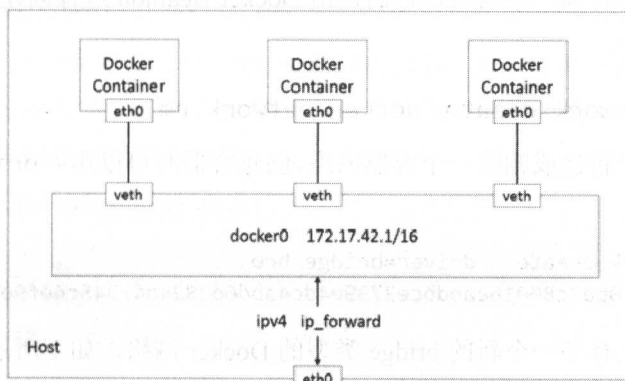


图 6-4 Docker 的默认网桥连接示意图

Docker 从其 1.9 版本开始引入了一整套的自定义网络命令和跨主机网络支持, 这是

libnetwork 项目从 Docker 的主仓库抽离后的又一次大动作，它将容器网络的控制能力更大程度开放给了终端用户，因此改变了连接两个容器所需的操作方式。这些与网络相关的操作被放到 `docker network` 命令中，并通过不同的子命令执行不同的具体行为，如下所示。

```
$ docker network --help
...
Commands:
  connect      Connect a container to a network
  create       Create a network
  disconnect   Disconnect a container from a network
  inspect      Display detailed information on one or more networks
  ls          List networks
  prune        Remove all unused networks
  rm           Remove one or more networks
```

下面简单介绍一下这些命令的作用。

1. `docker network ls`

此命令用于列出当前主机上的所有网络，这里列出的网络不仅仅是本地网络，还包括一些第三方插件所创建的跨节点网络，如下所示。

```
$ docker network ls
NETWORK ID   NAME      DRIVER    SCOPE
ac3437dd8382 bridge    bridge     local
c3d6ed6c765d host       host       local
c6c149d3ea88 none       null        local
```

在默认情况下会看到三个网络，它们是由 Docker Deamon 进程创建的，对应了 Docker 内置的三种网络插件。

2. `docker network create` / `docker network rm`

这两个命令用于新建或删除一个容器网络，创建容器时可以用 `--driver` 参数使用的网络插件，如下所示。

```
$ docker network create --driver=bridge br0
b6942f95d04ac2f0ba7c80016eabdbce3739e4dc4abd6d3824a47348c4ef9e54
```

现在这个主机上有了一个新的 bridge 类型的 Docker 网络，如下所示。

```
$ docker network ls
NETWORK ID   NAME      DRIVER    SCOPE
b6942f95d04a br0       bridge     local
... ..
```

Docker 容器可以在创建时通过 `--network` 参数指定所使用的网络，连接到同一个网络的容器可以直接相互通信。

当一个容器网络不再需要时，可以将它删除，如下所示。

```
$ docker network rm br0
```

3. docker network connect / docker network disconnect

这两个命令用于动态地将容器添加到一个已有网络，或将容器从网络中移除。为了比较清楚地说明这一点，下面来看一个例子。

参照前面的 libnetwork 容器网络模型示意图中的情形创建两个网络，如下所示。

```
$ docker network create --driver=bridge frontend  
$ docker network create --driver=bridge backend
```

然后运行三个容器，让第一个容器接入 `frontend` 网络，第二个容器同时接入两个网络，第三个容器只接入 `backend` 网络。首先用 `--network` 参数创建出第一和第三个容器，如下所示。

```
$ docker run -dt --name ins01 --network frontend alpine  
$ docker run -dt --name ins03 --network backend alpine
```

如何创建一个同时加入两个网络的容器呢？由于创建容器时的 `--network` 参数只能指定一个网络名称，因此需要在创建过后再用 `docker network connect` 命令将容器添加到另一个网络，如下所示。

```
$ docker run -td --name ins02 --network frontend alpine  
$ docker network connect backend ins02
```

在 Docker 的网络中内置有一个 DNS 域名解析服务，所有在同一网络中的容器都可以直接使用另一个容器的名字解析到它的 IP 地址，不在同一网络中的则不行。现在通过 `ping` 命令测试一下这几个容器之间的连通性，如下所示。

```
$ docker exec -it ins01 ping ins02  
可以连通  
$ docker exec -it ins01 ping ins03  
找不到名称为 ins03 的容器  
$ docker exec -it ins02 ping ins01  
可以连通  
$ docker exec -it ins02 ping ins03  
可以连通  
$ docker exec -it ins03 ping ins01  
找不到名称为 ins01 的容器
```



```
$ docker exec -it ins03 ping ins02
```

可以连通

使用容器的 IP 地址做这个实验会得到同样的结果。这个实验证实了在相同网络中的两个容器可以直接相互连接到对方，而不同网络中的容器是无法直接通信的。使用 `docker network disconnect` 命令可以动态地将指定容器从指定的网络中移除，如下所示。

```
$ docker network disconnect backend ins02
$ docker exec -it ins02 ping ins03
```

找不到名称为 ins03 的容器

可见，将 ins02 容器实例从 backend 网络中移除后，它就不能直接连通 ins03 容器实例了。

4. docker network inspect

这个命令可以用来显示指定容器网络的信息以及所有连接到这个网络中的容器列表，如下所示。

```
$ docker network inspect bridge
[{"Name": "bridge",
  "Id": "6e6edc3eee42722df8f1811cfd76d7521141915b34303aa735a66a6dc2c853a3",
  "Scope": "local",
  "Driver": "bridge",
  "IPAM": {
    "Driver": "default",
    "Config": [{"Subnet": "172.17.0.0/16"}]
  },
  "Containers": {
    "3d77201aa050af6ec8c138d31af6fc6ed05964c71950f274515ceca633a80773": {
      "EndpointID": "0751ceac4cce72...eb534184a4",
      "MacAddress": "02:42:ac:11:00:02",
      "IPv4Address": "172.17.0.2/16",
      "IPv6Address": ""
    }
  }
},
... ..]
```

值得指出的是，同一主机上的每个不同网络分别拥有不同的网络地址段，因此同时属于多个网络的容器会有多个虚拟网卡和多个 IP 地址。

5. docker network prune

这个命令会清理所有没有被任何容器使用的网络（不包括 Docker 默认创建的三个网络）。它通常用于定期清理人为创建后未被及时删除的网络资源，如下所示。

```
$ docker network prune
WARNING! This will remove all networks not used by at least one container.
Are you sure you want to continue? [y/N]
```

使用这个命令前请确保所有空闲的网络都不需要再被使用了。

从用户的角度来说，`docker network` 命令带来的最直观变化实际是：`docker0` 不再是唯一的容器网络了，用户可以创建任意多个与 `docker0` 相似的网络来隔离容器之间的通信。然而值得指出的是，用户自定义的网络和默认网络还是有不一样的地方。

首先，默认的三个网络是不能被删除的，而用户自定义的网络可以用 `docker network rm` 命令删掉。其次，连接到默认的 Bridge 网络的容器需要明确地在启动时使用 `--link` 参数相互指定，才能在容器里使用容器名称连接到对方。而连接到自定义网络的容器，不需要任何配置就可以直接使用容器名连接到任意一个属于同一网络中的容器。这样的设计既方便了容器之间进行通信，又能有效限制通信范围，增加网络安全性。

6.1.3 跨节点网络

Docker 内置的跨节点网络模块是在其 1.9 版本随着网络功能插件化的更新一起出现的。在这个版本到来之前，社区中就已经有许多第三方的工具和方案尝试解决这个问题，例如 Calico、Pipeworks、Flannel、Weave 等。由于各个解决方案的设计思路各有千秋，加上 CNM 和 CNI 模型的分化，因此即使在 Docker 推出内置跨节点网络以后，这些第三方方案依然欣欣向荣地发展。

既然每个容器都可以通过容器本地的 Bridge 网络获得独立的 IP 地址，为什么还不能让两个容器跨节点进行网络通信呢？这主要有两点原因（以下两张配图出自笔者的一次社区分享 PPT，有兴趣的读者可以在 <http://dockone.io/article/2616> 找到话题的完整内容）。

第一个原因是，由于不同节点上的 Docker 服务相互不知道对方的存在，因此在分配网络地址时，就有可能出现集群里两个不同主机各自运行的 Docker 容器获得相同 IP 地址的情况。即使将这些局部网络直接连接在一起也无法正确地通信，如图 6-5 所示。

问题一：容器地址重复

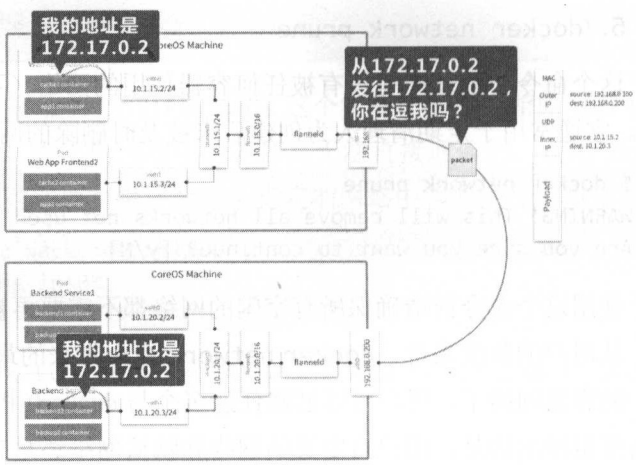


图 6-5 不同的节点获得了相同的 IP 地址

第二个原因是，即使所有节点的 IP 分配没有重复，各个节点上的主机路由表里也不包含通往其他节点上的容器地址的路由信息，因此发往跨节点容器地址的数据包还是会被丢弃，如图 6-6 所示。

问题二：容器地址不可达



在喵星好像没有“汪公馆”这个地址啊!!

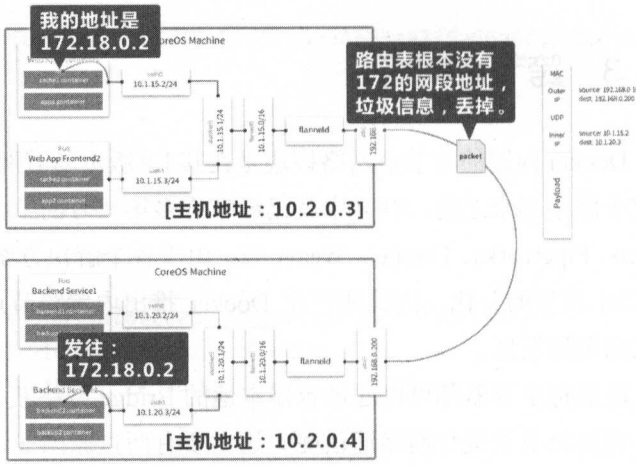


图 6-6 不同的节点上没有对方容器地址的路由信息

因此只要处理好这两个问题，跨节点的容器就可以各自拿着对方的 IP 地址进行通信了。IP 地址分配问题的解决思路相对清晰，既然各自分配无法保证唯一性，那么就需要对集群的 IP 采用统一分配。不过倘若真的设计一个系统来存储整个集群中成千上万的 IP 地址记录，每次进行路由的时候就都要到这个系统里查询一次目标，这显然是行不通的。现

实中的方案一般都使用分而治之的思路，即中心地址分配器只给每个主机分配一个固定的 IP 地址段，让主机在相应地址段内自己决定运行每个容器的 IP 地址。这种做法既能避免 IP 分配冲突，又能以极小的代价实现容器 IP 路由。具体来说，每个主机会知道到达当前节点全部容器的路由规则，以及集群所有 IP 段到对应主机的路由规则。路由过程分两段，第一段从源容器开始，由主机上的网络服务（使用额外封装改写目标地址为目标主机或用路由表锁定目标主机路由，详见后文）将数据包发送到目标容器所在的主机节点，第二阶段使用目标主机的本地路由规则将数据包送达到真正的目标容器。

乍看起来，这个中心地址分配器的设计并不复杂，不过考虑到集群稳定性，它不能是一个单点系统，最好还能在短暂网络分区（即部分节点与其他节点之间的网络断连，形成两个或多个小的子网络）的状况下避免出现数据不一致的情况。因此许多系统都选择了强一致性的 Raft 协议来确保信息的可靠性，其中 Flannel 和 Calico 都通过 Etcd 服务存储网络分配信息，Weave 内置了 Raft 实现，使用各个节点上的 Agent 服务兼职做网络分配数据的存储，Romana 自己实现了独立的地址分配服务。也有些网络方案，比如 Pipeline 和 OVS，虽然推荐基于 IP 段的划分机制，但索性不做地址自动分配的功能，只提供的一些辅助命令，让用户自行决定如何分配地址，然后手动操作。

相比分配唯一 IP 地址，跨节点网络连通的问题解决起来需要一点专业技能。目前主流的各种方案在实现细节上都存在一些差异，但其具体思路主要分为两种：覆盖网络和扩展路由。下面来具体介绍下这两种思路。

1. 覆盖网络

覆盖网络在有些地方也被称为隧道网络，它的原理是在不改变现有网络基础设施的前提下，将一种处于 ISO 模型第二层的协议（即传输层协议）附加到另一种二层协议或更高层协议之上，通过中继设备或终端设备对这种额外的封装数据进行处理，从而实现扩展现有标准传输协议（如 TCP、UDP 协议）的目的。这种“利用一种网络协议来传输另一种网络协议”的过程就像是在现有的网络设施上额外覆盖了一层信息，也像是一条新增的虚拟隧道实现数据点到点的准确投递。

这个思路可以简单地理解为，既然在现有的网络设施里只认识从节点到节点的路由，那么可以把要发送到目标容器的数据包先加上一个额外的传输协议包头（需要是 ISO 网络模型的二层或以上协议），将传输目的地写成目的节点的地址。等数据在目的节点上接收后再进行解包，拿掉这个临时的包头，得到真实的容器目的地址，然后在目的节点上进行本地路由就可以把数据包发到目的容器了。从数据的发送方来看，中间这一段增加包头和去

除包头的过程是透明的，就像是它直接写了目的容器的 IP 地址，然后信息就被原样传输到了那个容器里。

不过，既然要传输的也是二层协议报文，为何要把它封装到现有的二层（或更上层）协议报文里呢？从原理上说，若直接通过二层协议联通整个集群中的所有容器，即构造一个平坦的二层虚拟网络，通过特定网络设备直接路由，实现容器点到点之间的通信，也能实现所有容器的互通互联。这种方法需要二层网络设备支持，可能无法用在一些公有云基础设施上，通用性略差，更不用谈跨网络架构或是跨云部署的情况，但这并不是主要的问题。从直观感觉上说，大二层网络避免了额外的覆盖层装包、拆包，在传输效率上肯定比覆盖网络好。然而毫无隔离的二层平坦网络最终将使得任意一个寻址广播数据都会在整个数据中心内传播，导致广播数据泛滥，大量消耗网络带宽，因而可用的数据带宽随着网络规模的增加而迅速地下降，严重的网络广播风暴甚至会使整个数据中心网络瘫痪。

为了解决这种广播风暴问题，虚拟局域网（VLAN）技术被创造出来，它可以算是早期覆盖网络的代表之作。这种技术在普通的 TCP/UDP 包的数据部分定义了一个额外的 VLAN 包头，从而将局域网设备从逻辑上划分成一个个网段，每个虚拟机局域网是一个广播域，通过交换机设备在不同的虚拟局域网之间转发数据包。虚拟网段的区分是通过在 VLAN 包头中的一个 12 比特位的 VLAN tag 字段实现的，由于标记字段长度的限制，早期虚拟局域网技术只能支持最多 4096 个虚拟网段，这会对网络中设备数量的规模造成一定限制。为了突破这个上限，以虚拟可扩展局域网（VxLAN）等为代表的一系列新的覆盖网络技术被创造出来，这些技术采用扩展的隔离标识位数（例如 VxLAN 的 VLAN tag 长度变成 24 比特位，能支持 2^{24} （约一千六百万）个虚拟网段），在必要时可将广播流量转化为组播流量。覆盖网络的实现方式可以有多种，其中 IETF（国际互联网工程任务组）制定了三种 Overlay 的实现标准，分别是虚拟可扩展局域网（VxLAN）、采用通用路由封装的网络虚拟化（NVGRE）和无状态传输协议（SST），其中支持 VxLAN 标准的厂商的实力最为雄厚，可以说是经典覆盖网络的事实标准。由于采用内核自动编解码以及交换机进行硬件级的处理转发，这些经典覆盖网络的传输效率并不差，通常可以保持原始二层网络至少 90% 以上的传输带宽^①。

但为什么在网络上有时会看到文章说容器的覆盖网络传输效率很低呢？这种认识其实是由于以 Weave 和 Flannel 等为代表的早期容器覆盖网络开始流行（大约是在 2014 年，也就是 Ubuntu 14.04 和 CentOS 6.x 发行版大行其道的时候）。不幸的是这些主流 Linux 发行版

^① <http://packetpushers.net/vxlan-udp-ip-ethernet-bandwidth-overheads/>

使用的内核版本都没有内置 VxLAN 等标准覆盖网络内核模块。为了让用户上手更加容易，这些工具选择了自行实现网络包头的封包、解包功能，即用户态的覆盖网络功能。由于网络包的发送和接收本来是在 Linux 的内核态中完成的，现在需要将这些数据包先传递到用户态，经过 Flannel 或 Weave 服务的加工处理，然后回到内核态继续传输，一装一拆间就增加了四次内核上下文切换。这样的覆盖网络完全不依赖于内核协议和其他网络设备，因此通用性特别强，所有的配置都可以在主机的服务上完成，使得即使是对底层网络一无所知的小白用户都可以快速实现跨节点的容器互联互通。但代价也是比较明显的，这类覆盖网络的实际带宽大约只有物理网络带宽的 20%，早期版本 Weave 实现的 UDP 覆盖网络效率甚至还不到物理带宽的 6%，远远低于 IETF 的标准协议^①，因而造成了早期用户存在容器覆盖网络效率低下的错误印象。不过不可否认，正是这种现在看来十分低效的覆盖网络，在当时为容器集群化运用的开展起到了重要的启蒙和普及作用。

常见的基于覆盖网络实现的开源跨节点容器网络解决方案有以下这些。

- Docker 内置的“Overlay”网络插件：基于 VxLAN 实现的高效易用的跨节点网络，仅支持 CNM 接口。
- Weave: WeaveWorks 公司提供的企业级覆盖网络方案，支持应用层封装的覆盖网络和基于 VxLAN 的覆盖网络，支持 CNM 和 CNI 网络模型。
- Flannel 的 UDP 和 VxLAN 模式：CoreOS 公司的通用覆盖网络解决方案，支持应用层封装和 VxLAN 封装的覆盖率，也支持基于扩展路由的网络连通方法，支持 CNM 网络模型。
- Pipework: 其实是一套复杂的 Bash 脚本工具，可以简化用户建立基于 VxLAN 等容器网络结构的过程。
- OVS (Open vSwitch): 开源的网络即服务解决方案，可以基于 VxLAN 和 GRE 协议进行容器跨节点路由。
- 开启 IPIP 模式的 Calico: 这是一种将 IP 包封装在 IP 数据包内，目的是为了在无法使用 BGP 协议的网络里实现分段路由，从而将数据包跨节点投递到目标容器的 IP 地址，支持 CNM 和 CNI 网络模型。

2. 扩展路由

跨节点网络的另一种实现思路是扩展路由。这种思路的原理是通过某种低成本的机制让容器的 IP 地址和主机的 IP 地址一样，可以直接在集群的网络基础设施上进行路由，从

^① <http://www.generictestdomain.net/post/weave-is-kind-slow/>

而解决容器跨节点通信问题。它们的共同特点是：在网络传输过程中没有额外的封装、没有 NAT 地址转发，因此性能普遍很好，接近原始网络带宽，并且由于传输过程使用的都是标准网络协议，在出现丢包等网络问题时，可以利用传统的网络工具进行故障分析。扩展路由的具体实现方法有很多，目前有较多实际应用的主要有三种：Macvlan、节点网关路由和节点 BGP 路由。

Macvlan 技术能够在主机的网卡上添加多个 Mac 地址（实质是虚拟出多个子网卡，如 eth0.1、eth0.2……），从外界看来，就像是把网线分成两股，分别接到了不同的主机上一样。单纯看这个功能也许说明不了什么，不过若是在主机的外部网卡上创建多个子网卡，让它们分别绑定 IP 地址和 Mac 地址，再将每个子网卡分配给不同的 Network Namespace，这就相当于把所有容器的网络都直连到主机网卡上，使得每一个容器都有一个外部可访问的 Mac 地址和 IP 地址，基于 Macvlan 的网络可以让容器使用与传统虚拟机一样的成熟技术进行数据路由。此外，Macvlan 本身支持划分虚拟局域网，因此可以避免广播风暴的发生。

使用 Macvlan 需要集群中的主机使用至少 3.9 版本以上的 Linux 内核，且使用前可能需要先手动加载相应的系统内核模块，如下所示。

```
$ modprobe macvlan
$ lsmod | grep macvlan
macvlan 19046 0
```

如果第一个命令报错，或者第二个命令没有显示已加载的 Macvlan 模块信息，则说明当前系统的内核不支持 Macvlan 功能。除了对节点系统的要求，这个方案还需要网络中的所有二层交换设备也具备 Macvlan 特性的支持，事实上目前绝大多数的公有云平台都无法使用 Macvlan 方案。

节点网关路由和节点 BGP 路由这两种方案的思路比较相似，它们都是在主机的路由表上做文章。用简单的方式来理解，既然跨主机的容器之间无法路由是因为它们相互不知道对方的路由方式，那么只要把这些容器 IP 段到目标主机的路由信息添加到路由表里，不就完成了从源容器到目标主机的第一段路由了吗？只要数据包到了目标主机，再到目标容器就是本地路由能够直接完成的事情了。

沿着这个思路，可以想到的一种实用可行的办法就是在每个节点上运行一个 Agent 服务，然后用这个 Agent 监听容器 IP 段分配的变化，一旦有变化发生就将新的路由规则刷到本地的内核路由表里。运行在每个节点上的 Agent 就是一个控制网络联通规则的网关程序，因此这种方法被称为“主机网关”。这样看来似乎只要确保路由表准确、及时，路由问题就被轻松解决了。其实在二层网络的环境里，这是没问题的，然而一旦节点之间隔着另一个

路由器，情况就没那么顺利了。由于 Agent 服务只存在于主机，但数据经过网络上的路由器时，这个路由器没有相应的路由信息，就会将数据包丢弃。前面已经讲过，由于网络风暴的存在，传统的二层网络的规模通常都很小，需要采用三层设备（比如路由器）来划分子网、控制寻址包的广播范围。随着技术的发展，也出现了一些特殊的二层设备，能够通过避免环状路由线路和控制广播转发距离等手段在较大范围上解决广播风暴的发生，从而实现容纳几万甚至更大规模的二层网络，被称为“大二层网络”。但目前来说，这种技术的普及度并不高，并且它也不适用于与分布位置较远的跨机房网络连接。

主机网关路由无法跨越三层网络的原因在于，它无法改变在三层网络上其他路由器设备的路由规则。那么是否有一种方式能够做到这点呢？

BGP（Border Gateway Protocol，边界网关协议）是运行于 TCP 上的一种基于自治系统的路由协议。它是目前唯一能处理因特网规模的网络路由协议，因此广泛存在于各种基础设施网络里。简单来说，节点 BGP 路由网络的思路就是让安放在各个节点上的 Agent 不仅会修改本地内核路由表，还能实现路由 BGP 协议，把自己变成容器所在 IP 子网段的末端路由器，利用 BGP 协议具有的路由信息传播机制，让网络里的其他三层设备学习到容器网段的路由信息，这就是节点 BGP 路由方案。

在节点 BGP 路由方案的网络里，一个节点上的容器发送数据包给网络中另一个节点上的容器时，这个数据包首先会在当前节点的 Linux 内核中进行一次网络路由，接着通过集群网络设施传送到目标节点，然后在目标节点上又进行一次路由，最终到达目的容器。就像网络中的两个虚拟机在通信，只不过在现有网络连接的基础上增加了一前一后两个路由器。这种方案是架构在三层网络之上的，因此不会产生广播风暴的风险。不过对于规模比较大的网络，为了避免在网络的所有路由设备中产生庞大的路由表，可能需要人为划分路由自治区域。BGP 可以通过它的两个子协议，IBGP（Internal Border Gateway Protocol）和 EBGp（External Border Gateway Protocol）划分子路由域。目前主流的路由器设备都提供 BGP 协议支持，因此在绝大多数公有云上均可以实施节点 BGP 方案，但在有些企业内网环境里，依然可能出现基础网络设施不支持 BGP 协议的情况（通常是因为网管关闭了该特性），此时这种方案就无法使用了。Calico 是目前最主流的企业级节点 BGP 路由容器网络产品。

常见的基于扩展路由实现的开源跨节点容器网络解决方案有以下这些。

- Docker 内置的“Macvlan”网络插件：开箱即用的 Macvlan 网络，仅支持 CNM 接口。
- Pipework：使用这套脚本工具同样可以便于用户配置基于 Macvlan 的网络。

- Calico（非 IPIP 模式）：基于节点 BGP 协议的路由的方案，支持很细致的 ACL 网络访问规则控制。
- Romana：基于内核路由表和 Iptables 规则路由的方案，同样支持细致的 ACL 网络访问规则控制^①。
- Flannel 的 HostGW 模式：基于内核路由表和 Iptables 规则路由的方案。

6.1.4 使用 Docker 内置的 Overlay 类型网络

在之前介绍 `docker network ls` 命令时，如果注意观察就会发现，默认的网络以及自定义的 Bridge 类型网络在 SCOPE 一栏的输出值均为“local”。这个类型表示该网络仅仅存在于当前节点，因此网络中的容器也只能在当前节点内部进行通信。Docker 内置了 overlay 和 macvlan 两种跨节点的 CNM 网络驱动，它们需要在当前节点使用 `docker swarm` 命令加入某个 SwarmKit 集群才能被启用。

Docker 创建 SwarmKit 集群的命令已在第 2 章关于 Swarm Mode 的小节里介绍过，这里假设有两个节点，通过以下命令建立了一个小型集群。

```
$ docker swarm init                # 节点 1
$ docker swarm join --token ..... # 节点 2
```

以使用 overlay 网络为例，创建一个跨节点的容器网络，如下所示。

```
$ docker network create o1 --driver overlay
$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
bdwqdc1ro6bc	o1	overlay	swarm
ac3437dd8382	bridge	bridge	local
c3d6ed6c765d	host	host	local
c6c149d3ea88	none	null	local

这个使用 Overlay 网络驱动创建的网络，其 Scope 属性值被设置成了 swarm，意味着它能够在整个 SwarmKit 集群里被使用。此时如果到集群的另一个节点里执行 `docker network ls` 命令，同样会看到这个名称是“o1”的网络。仔细观察后会发现，在这两个节点上，所有 Scope 为 local 的同名网络都具有不同的网络 ID 值，而 Scope 为 swarm 的同名网络则具有相同的网络 ID。这间接说明了前一种网络只属于特定节点，后一种网络属于整个集群。

其实可以再创建一个 Overlay 类型的网络，然后在这个集群里重复之前的网络连通性

^① http://romana.io/how/romana_details/#romana-tenant-isolation

实验，如下所示。

```
$ docker network create o2 --driver overlay
```

在节点 1 创建两个容器，分别加入 o1 和 o2 网络，如下所示。

```
$ docker run -dt --network o1 --name c1 alpine
$ docker run -dt --network o2 --name c2 alpine
```

在节点 2 创建一个容器，加入 o2 网络，如下所示。

```
$ docker run -dt --network o2 --name c3 alpine
```

然后从 c2 容器里分别尝试访问 c1 和 c3，如下所示。

```
$ docker exec c2 ping c1
ping: unknown host

$ docker exec c2 ping c3
PING c3 (172.18.0.3): 56 data bytes
64 bytes from 172.18.0.3: icmp_seq=0 ttl=64 time=0.205 ms
```

可以看到，虽然 c2 和 c1 容器在同一个主机上，但由于不属于同一个网络，它们之间无法通信。而 c2 和 c3 在两个不同的主机上，它们之间却可以直接连接。可见在使用了 Overlay 类型驱动的 Docker 网络中，容器之间是否可连接完全由它们是否处于同一网络来决定，而与它们是否处于同一主机节点无关。

6.1.5 构建基于 Flannel 的覆盖网络

Flannel 提供了应用层封装和 VxLAN 封装的覆盖网络支持，它提供基于节点网关路由的扩展路由网络支持。由于 Flannel 诞生于 CNM 和 CNI 标准出现以前，因此可以不依赖于特定的标准接口实现容器的互联（但也可以通过 CNI 接口提供服务），具有很强的通用性。

这里主要介绍 Flannel 的覆盖网络方案，并以传统的应用层封装模式为例解释整个通信的过程。图 6-7 展示了这种网络的工作原理。

数据从一个容器发往运行在另一个物理节点上的指定的容器地址时，首先经过容器自身 Network Namespace 内部的网卡，这个网卡和主机网桥（如 docker0）通过 Veth Pair 相连。然后根据目的 IP 地址的路由，数据会从主机网桥被发往主机上的另一个网桥 flannel0，从而被守护进程 flanneld 接收处理。flanneld 会将收到的数据再次打包，然后根据 Etcd 上的配置记录，通过 UDP 协议发送到相应目标主机 IP 地址的 UDP/8285 端口，并由这个主机的

flanneld 守护进程接收并解包。最后，这个被还原了的原始数据包依次经过目的主机的 flannel0 和主机网桥（比如 docker0）转发，到达目的容器内部。

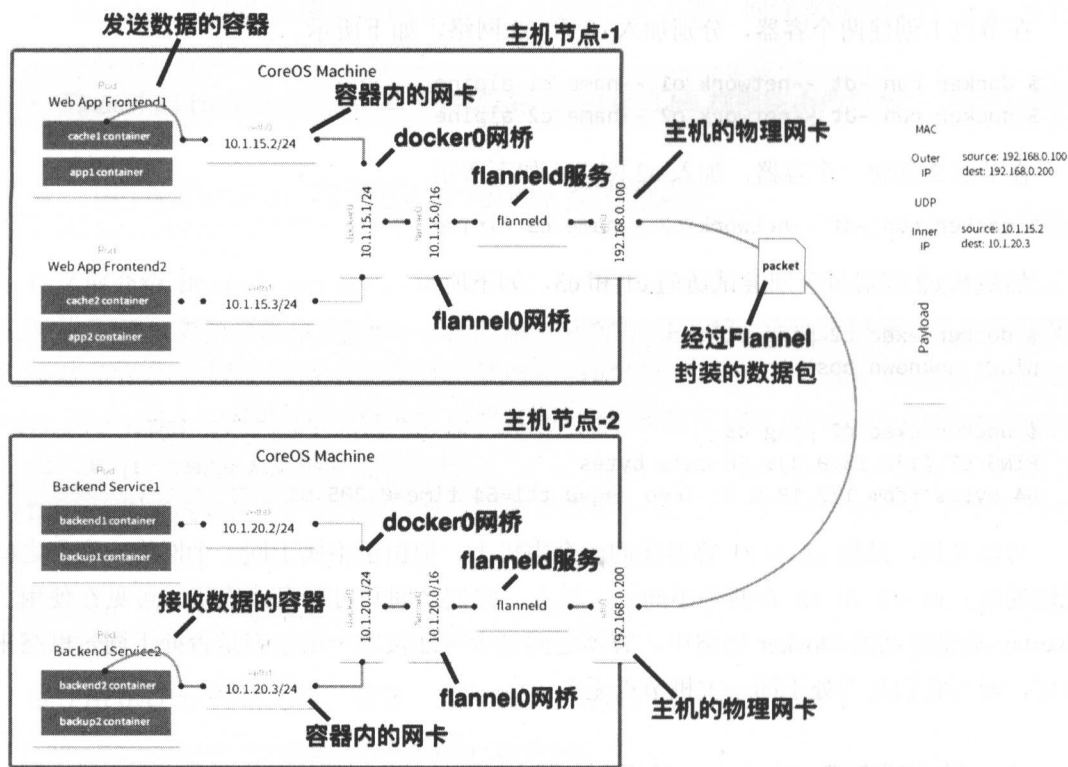


图 6-7 Flannel 节点间的容器通信示意图

这个流程与普通网络通信最显著的不同点就在于 Flannel 服务对通信数据进行了特殊的打包和转发。由于待发送的实际数据内容是被封装后通过 UDP 协议发送到目的节点的，Flannel 的这种在应用层进行封装、解包的覆盖网络在许多地方也被称为基于 UDP 转发的覆盖网络。

以一个 ping 命令为例，此时如果在主机上使用 tcpdump 或 sysdig 命令抓取所有网络通信数据，然后在 WireShark 中进行分析，就会发现如图 6-8 所示的这种数据包。

在这个通信记录中，可以在 UDP/8285 端口上看到有两个序号连续但方向相反的数据包。这两个包的 UDP 数据部分实际上分别封装了一次 ping 命令的 ICMP 请求和应答内容。

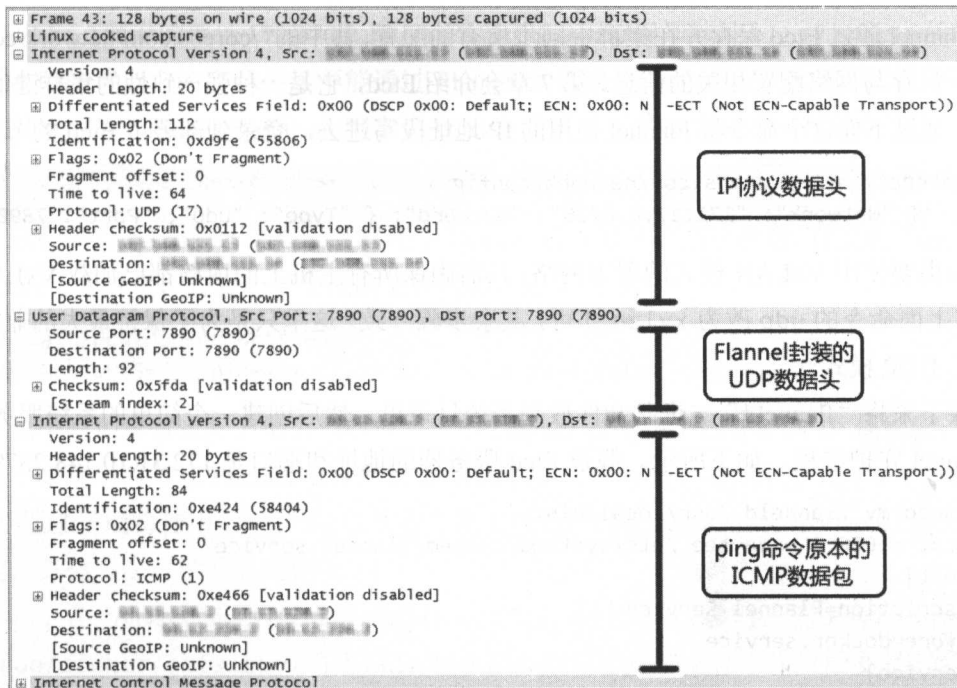


图 6-8 通过 Wireshark 解析出的 UDP 封装后的 ping 命令

从 GitHub 下载编译好的最新 Flannel 程序。解压缩后会得到一个可执行文件“flanneld”和一个脚本文件“mk-docker-opts.sh”，如下所示。

```
$ wget https://github.com/coreos/flannel/releases/download/v0.8.0/
flannel-v0.8.0-linux-amd64.tar.gz
...
$ tar xzf flannel-v0.8.0-linux-amd64.tar.gz
$ ls
README.md flanneld mk-docker-opts.sh
```

然后需要为 Docker 的网络重新选择一段 IP 地址区域，可以使用任意未被占用的内网 IP 段。TCP/IP 协议中预留了三段内网专用的地址，如下所示。

- A 类地址：10.0.0.0 ~ 10.255.255.255。
- B 类地址：172.16.0.0 ~ 172.31.255.255。
- C 类地址：192.168.0.0 ~ 192.168.255.255。

下面例子使用 172.17.0.0/16 作为 Docker 网络的地址区域。这个地址段应该被指配给 Flannel，再由 Flannel 从这个地址段区域中分配一个子区域给每个服务节点上的 Docker 服务进程，从而确保运行在不同节点上的 Docker 容器一定会获得互不重合的 IP 地址。

Flannel 通过 Etcd 保存并在集群中同步所有的配置，其中的“/coreos.com/network/config”键用于保存与网络配置相关的信息。第 7 章会介绍 Etcd，它是一种强一致性的集群键值存储服务，通过下面这个命令将 Flannel 使用的 IP 地址段写进去，登录到部署了 Etcd 的节点上。

```
$ etcdctl set /coreos.com/network/config \
  '{ "Network": "172.17.0.0/16", "Backend": { "Type": "udp", "Port": 7890 } }'
```

如果要使用 VxLAN 模式的覆盖网络，只需确保所有主机上的内核都已加载 VxLAN 模块，将上面命令的 `udp` 改为 `xvlan` 即可，其余步骤一致。这里只是为了保持最大的兼容性，采用了 UDP 模式。

接下来将“flanneld”二进制文件放到系统目录里，然后创建一个简单的系统服务来启动 Flannel 守护进程，如下所示。假设 Etcd 服务器的地址和端口是 172.31.10.148:2379。

```
$ sudo mv flanneld /usr/local/bin/
$ cat <<EOF | sudo tee /etc/systemd/system/flannel.service
[Unit]
Description=Flannel Service
Before=docker.service
[Service]
ExecStart=/usr/local/bin/flanneld --ip-masq=true
--etcd-endpoints=http://172.31.10.148:2379
[Install]
WantedBy=multi-user.target
EOF
$ sudo systemctl start flannel
```

Flannel 会自动建立一个新的虚拟网卡 `flannel0`，并根据 Etcd 中的配置为它自动选择一个指定区域内的网卡 IP 地址。通过 `ip` 命令能够看到，在这个例子中，当前节点分配到的地址是 172.17.64.0/16，如下所示。

```
$ ip addr show flannel0
4: flannel0@NONE: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> ...
... ..
    inet 172.17.64.0/16 scope global flannel0
```

现在虽然 Flannel 服务已经启动了，然而节点上的 Docker 服务并没有接受 Flannel 的管理，这一点可以从 `docker0` 的网卡地址看出来，如下所示。

```
$ ip addr show docker0
3: docker0@NONE: <BROADCAST,MULTICAST,UP,LOWER_UP> ...
... ..
    inet 172.17.42.1/16 scope global docker0
```

docker0 网卡的地址 172.17.42.1/16 并不属于 flannel0 网卡地址 172.17.64.0/16 的一部分, 此时不同的节点上的容器仍然不能通过 Flannel 的网络进行通信。为此需要运行 Flannel 安装包目录中的 mk-docker-opts.sh 脚本为 Docker 生成必要的配置, 如下所示。

```
$ sudo ./mk-docker-opts.sh -c
```

运行完后, 这个脚本会自动生成用于 Docker 后台服务启动的参数, 存放到“/run/docker_opts.env”文件里, 查看一下生成的文件内容, 如下所示。

```
$ cat /run/docker_opts.env
DOCKER_OPTS="--bip=172.17.71.1/24 --ip-masq=false --mtu=1472"
```

这个附加参数实际上是为当前节点上的 Docker 指定一个预分配的 IP 地址段, 这个地址段一定是当前节点 Flannel 服务所属 IP 地址段的一个子段。在默认情况下, Flannel 分配的 IP 地址子段将使用 24 位网络地址和 8 位主机地址, 这个值可以通过 Flannel 的配置进行设定, 具体方法可参看其文档^①。

然后在“/lib/systemd/system/docker.service”文件 Service 区域加上一行 EnvironmentFile =-/run/docker_opts.env, 再重启 Docker 服务以启用这个文件中设置的变量, 如下所示。

```
$ sudo sed -i '/\[Service\]/ aEnvironmentFile=-/run/docker_opts.env' \
/lib/systemd/system/docker.service
$ sudo sed -i 's#/usr/bin/dockerd#/usr/bin/dockerd $DOCKER_OPTS#' \
/lib/systemd/system/docker.service
$ sudo systemctl daemon-reload
$ sudo systemctl restart docker
```

这样就完成了一个节点上的 Flannel 安装和部署。最后, 用同样的方法将 Flannel 启动到每个节点上, 整个 Flannel 网络的架设就完成了。

此时, 如果观察各个节点的 flannel0 和 docker0 网卡获得 IP 地址, 会发现它们都同属于 172.17.0.0/16 地址段中的一个子区域。事实上, Flannel 将所有子网分配的信息都保存在了 Etcd 的“/coreos.com/network/subnets”目录中, 如下所示。

```
$ etcdctl ls /coreos.com/network/subnets
/coreos.com/network/subnets/172.17.43.0-24
/coreos.com/network/subnets/172.17.64.0-24
/coreos.com/network/subnets/172.17.83.0-24
```

查看其中任意一个键的内容, 就是这个 IP 段对应的实际服务节点路由地址, 如下所示。

^① <https://github.com/coreos/flannel/blob/master/Documentation/configuration.md>

```
$ etcdctl get /coreos.com/network/subnets/172.17.64.0-24
{"PublicIP": "172.31.14.97"}
```

现在如果在集群的两个主机节点上各启动一个 Docker 容器，然后直接使用 ping 命令连接对方的容器 IP 地址，就会发现容器的网络已经能够互通了。第一个节点如下所示。

```
$ docker run -dt --name workA alpine
```

另一个节点如下所示。

```
$ docker run -dt --name workB alpine
$ docker exec workB ping workA
PING workA (10.1.192.0): 56 data bytes
64 bytes from 10.1.192.0: seq=0 ttl=64 time=0.072 ms
```

如果采用 CNI 插件的方法部署 Flannel 则会简单很多，但理解手动部署 Flannel 的过程更有助于了解它和覆盖网络的工作原理。

6.1.6 构建基于 Calico 的 BGP 路由网络

Calico 通过 BPG 协议和每个节点内核实现的路由转发机制，使得整个 Calico 网络中所有的节点在 IP 层互联，充分利用数据中心的网络结构，不依赖特殊硬件，整个过程也没有额外的 NAT 或数据封装。典型的 Calico 网络由几个组件构成，如图 6-9 所示。

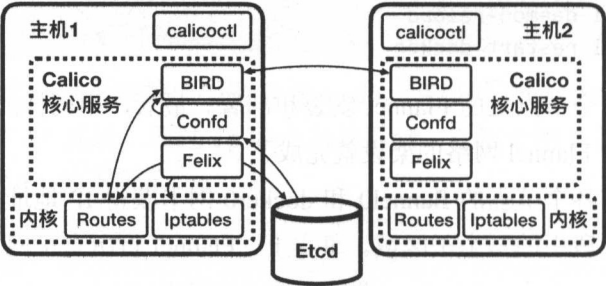


图 6-9 Calico 的主要组件

- Felix: Calico 后台服务，需要运行在集群的每个节点上，负责配置路由及网络访问规则，确保节点之间的路由正确。
- Etcd: 强一致性的分布式键值存储服务，负责网络元数据的保存，确保 Calico 网络配置信息的全局唯一。
- BIRD (BGP Client): BGP 协议后台服务，负责把 Felix 生成的路由变更信息分发到

整个 Calico 网络，确保各节点间的路由信息能及时更新。

- **BGP Route Reflector:** BIRD 服务的替代品，只在大规模部署时需要用到。相比 BIRD 服务所有节点互联的路由信息分发方式，它采用了集中式路由管理，通过一个或多个 BGP Route Reflector 服务来完成集中式的路由分发。
- **Confd:** 用于监听 Etcd 存储的配置信息变化，并通过模板自动生成 BIRD 服务配置文件。

Calico 提供了适配 CNM 和 CNI 的相应实现，可以用于原生 Docker 集群、Kubernetes 集群、Mesos 集群等，还可以结合 Openstack 的 Neutron 组件用于虚拟机网络。下面以使用原生的 Docker 集群为例，介绍具体的 Calico 服务部署过程。

按照 CNM 的标准，提供网络功能的第三方插件需要有一个用来处理网络请求的后台服务。为了方便使用，Calico 已经将这个服务做成了容器的镜像，并提供一个 `calicoctl` 工具来简化与 Calico 服务相关的用户操作。

首先从 GitHub 下载最新的 `calicoctl` 工具二进制文件，并赋予可执行权限，如下所示。

```
$ sudo wget -O /usr/local/bin/calicoctl https://github.com/projectcalico/calicoctl/releases/download/v1.2.1/calicoctl
$ sudo chmod +x /usr/local/bin/calicoctl
```

Calico 将网络中的所有节点、IP 池、网络接口等都抽象为“资源”进行关联，这种做法与 Kubernetes 有异曲同工之妙。`calicoctl` 工具包含了许多子命令，如下所示。

```
$ calicoctl --help
...
  create      从标准输入或者文件创建资源
  replace     从标准输入或者文件更新资源
  apply       相当于上面两个命令的结合，若资源不存在则创建，否则更新
  delete      从文件、标准输出或者资源类型和名字删除资源
  get         通过文件、标准输入或者资源类型和名字获取定义的资源
  config      管理系统层和较低级别的节点配置选项
  ipam        IP 地址管理
  node        Calico 节点管理
```

为了确保网络元信息的高可用性，Calico 使用 Etcd 保存它的运行时配置。因此需要用用户预先部署 Etcd 服务，然后（在每个节点上）将 Etcd 服务的地址写入到 Calico 的配置文件里。假设 Etcd 运行在 172.31.10.148 服务器的 2379 端口上，如下所示。

```
$ sudo mkdir /etc/calico
$ cat <<EOF | sudo tee /etc/calico/calicoctl.cfg
apiVersion: v1
```

```
kind: calicoApiConfig
metadata:
spec:
  etcdEndpoints: http://172.31.10.148:2379
EOF
```

为了让集群中的所有节点在不启动 Swarm Mode 的情况下共享 Docker 网络配置，必须配置使用外部存储保存容器的配置信息。在每个节点上修改 Dockerd 服务的描述文件，添加--cluster-store 参数，如下所示。

```
$ sudo sed -i 's# /dockerd#/dockerd --cluster-store=etcd://172.31.10.148:2379#' \
  /lib/systemd/system/docker.service
$ sudo systemctl daemon-reload
$ sudo systemctl restart docker
```

然后使用 calicoctl 工具在每个节点启动 Calico 的后台服务，如下所示。

```
$ sudo calicoctl node run
... ..
Starting libnetwork service
Calico node started successfully
```

这个命令会自动从网络下载相应的 Docker 镜像，稍等几分钟后，使用 docker ps 就能看到有一个名称是“calico-node”的容器在节点上运行起来了，如下所示。

```
$ docker ps -a
CONTAINER ID   IMAGE                                ... .. NAMES
c34c653fdb8a   quay.io/calico/node:v1.2.1         ... .. calico-node
```

使用 calicoctl 工具可以查看 Calico 后台服务的运行状态和当前介绍的 Calico 网络路由信息，如下所示。

```
$ sudo calicoctl node status
Calico process is running.
```

IPv4 BGP status

PEER ADDRESS	PEER TYPE	STATE	...	INFO
172.31.8.148	node-to-node mesh	up	...	Established
172.31.7.182	node-to-node mesh	up	...	Established

查看 Calico 给容器预留的 IP 池，如下所示。

```
$ calicoctl get ippool -o wide
CIDR          NAT    IPIP
192.168.0.0/16 true   false
fd80:24e2:f998:72d6::/64 true   false
```

这个默认的 C 类地址 IP 池中的地址段很容易与内网其他主机地址冲突,可能需要将它删掉,然后重新创建一块与内网主机不重合的 IP 段。在 Calico 中 IP 池是一种资源,可以使用 YAML 文件描述,如下所示。

```
$ calicoctl delete ippool 192.168.0.0/16
Successfully deleted 1 'ipPool' resource(s)
```

```
cat <<EOF >ippool.yml
apiVersion: v1
kind: ipPool
metadata:
  cidr: 10.1.0.0/16
spec:
  ipip:
    enabled: false
    nat-outgoing: true
    disabled: false
EOF
```

```
$ calicoctl create -f ippool.yml
Successfully created 1 'ipPool' resource(s)
```

```
$ calicoctl get ippool -o wide
CIDR          NAT    IPIP
10.1.0.0/16   true   false
fd80:24e2:f998:72d6::/64 true   false
```

如果是在云环境(譬如 AWS、阿里云等)部署, BGP 协议很可能用不了,这种时候如果使用 Calico 方案,可以降级为 IPIP 方式的覆盖网络,只需将刚刚资源描述文件中的 `spec.ipip.enabled` 设置成 `true`。由于 IPIP 协议在较新的内核里已经内置,这种方式的传输效率与 Flannel 的 VxLAN 模式相当。

有了合适的 IP 池以后,就可以在 Docker 中使用 `docker network create` 创建一个基于 Calico 的跨容器网络,如下所示。

```
$ docker network create --driver=calico \
  --ipam-driver=calico-ipam --subnet=10.1.0.0/16 cali01

$ docker network ls
```

```
NETWORK ID   NAME      DRIVER  SCOPE
a357d258c2b8 cali01    calico   global
... ..
```

在不同节点上分别创建一个容器，加入“net1”网络。其中一个节点如下所示。

```
$ docker run -dt --network cali01 --name workA alpine
```

另一个节点如下所示。

```
$ docker run -dt --network cali01 --name workB alpine
$ docker exec workB ping workA
PING workA (10.1.192.0): 56 data bytes
64 bytes from 10.1.192.0: seq=0 ttl=64 time=0.072 ms
```

Calico 基于内核的 Iptables 提供了丰富而灵活的网络访问控制规则（ACL），可以实现节点的多租户网络隔离、安全组以及其他可达性限制的功能。在 Calico 中访问控制规则同样被作为一类资源进行管理，并通过 YAML 文件描述和创建。这里不再详细展开，具体操作可参看其文档^①。

6.2 容器存储

6.2.1 容器实例和镜像的存储

容器在存储方面的实现包含两个差异很大的部分，一部分是容器实例和镜像的存储，另一部分是数据卷的存储。在这个小节里先简单讨论一下容器是如何存储实例和镜像的。

容器的镜像格式是有规范的，在这方面，由“开放容器计划组织（Open Container Initiative）”发起的 OCI 应用容器规范已经成为了所有主流容器遵循的现实标准，Docker 镜像的结构就是这个规范体系的一种实现。其他还有点影响力的要数 CoreOS 公司曾经提出的 AppC 容器规范了，它制定了一种不太一样的容器镜像格式，不过只有 Rkt、Kurma、Jetpack 等少数容器项目真正实现过该规范。

在 OCI 规范里，容器镜像的结构是分层的，这样可以最大化地利用本地磁盘的存储空间。由于大多数的容器镜像都会基于一些标准的基础镜像来构建，在它们中有很长的一部

^① <http://docs.projectcalico.org/v2.2/getting-started/docker/tutorials/security-using-calico-profiles-and-policy>

分文件内容完全相同，采用分层的存储结构有利于让不同镜像复用这些相同的部分。此外容器运行时，相当于在已有镜像的基础上增加或删除内容，分层结构还能让通过同一个镜像创建的容器以只读数据层的方式复用镜像的内容，从而避免每个容器全量复制镜像内容带来的开销。图 6-10 是一张很经典的介绍容器实例运行时的存储结构的图，图中的两个容器各自采用只读的方式复用了镜像的每层数据，并且在顶层加上一个可写入层，所有在容器中所做的文件内容修改都以增量存储的方式记录在这一层里。倘若节点上有多个相同镜像的容器在运行，它们会共享一份镜像数据的磁盘空间，加上各自在可写入层基于原始镜像修改过的很少的一部分文件的磁盘空间。

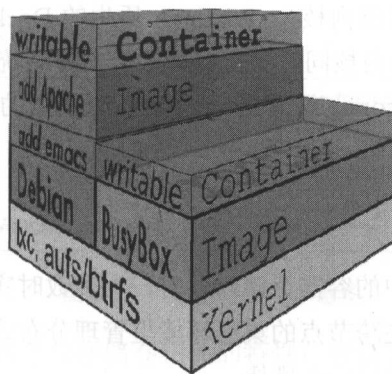


图 6-10 容器运行时的分层结构

为了实现这种可复用的分层存储结构，Docker 采用了一些特殊的文件系统结构。AUFS 是 Docker 最早支持的一种镜像和容器实例存储实现，AUFS 全称是 AnotherUnionFS（另一个联合文件系统），它的前身是另一个名为 UnionFS（联合文件系统）的项目。所谓的“联合文件系统”是指将普通的文件系统内容合并来，形成一个单一的挂载点。实际上用户无法将一块磁盘直接格式化成 AUFS 或其他“联合文件系统”格式，从某种角度看，联合文件系统仅仅是将多个已经存在的目录进行合并的虚拟文件视图。许多联合文件系统都支持“写时复制（CoW）”的技术，任何对其中文件的修改都会被“向上拷贝”到文件系统的一个临时的分层里面，而不会改变影响构成该文件系统的原始数据内容。

现在 AUFS 早已不是 Docker 所支持的唯一联合文件系统类型。为了支持更多的容器存储种类，Docker 抽象了一层 Storage Driver 接口，任何一种联合文件系统只需实现相应的接口功能就可以为 Docker 提供容器运行时的存储支持，目前 Docker 内置的 Storage Driver 类型包括 vfs、aufs、overlay、overlay2、btrfs、zfs、devicemapper 和 windows 等。由于 OCI 采纳了分层镜像格式，使用该标准的容器实现大多会基于某些联合文件系统，只是支持的种类可能不及 Docker 丰富，例如 LXC 支持 aufs、overlayfs、btrfs、zfs 等文件系统，Rkt

的镜像运行机制主要是基于 overlayfs 实现的。

关于这些文件系统的细节原理和差异对比，在许多介绍 Docker 技术的书籍以及网络上都能找到很好的参考资料，鉴于本地容器和镜像的存储不是本书要讨论的重点问题，这里不再详细展开。

Docker 会根据操作系统的不同自动选择最合适的 Storage Driver 类型，例如在 Debian 和 Ubuntu 系统的 Linux 默认采用较稳定的 AUFS 文件系统，对于 CoreOS 系统或内核高于 4.0 版本的 RedHat、CentOS 系统默认会用 Overlay2 文件系统，而对于其他内核较低的 Linux 系统则默认使用兼容性较好的 DeviceMapper 驱动（这种文件系统驱动在效率和稳定性方面有较多问题，因此通常建议升级内核）。Windows 原生的 Docker 则会使用专用的 Windows 文件系统驱动。若当前系统的内核同时支持多种联合文件系统类型，可以在启动 Docker 的后台服务 dockerd 时通过 `--storage-driver` 参数指定所用的容器存储种类。

6.2.2 容器卷的存储

当人们在谈论集群环境中的容器存储问题时，大多数时实际在说容器的卷存储。和跨节点的网络通信一样，如何在跨节点的集群环境里管理分布式服务的数据存储，也是大规模使用容器技术时需要面临的另一个挑战。

卷存储（Volume）的概念在 Docker 很早的版本中就已存在，一个数据卷相当于一个从外部挂载到容器中的目录，所有写到这个目录里的内容都会被记录到数据卷所指定的位置进行持久化存储。

这样做有几个主要原因。首先，特定容器的生命周期可能比它产生的数据文件所需的生命周期更短，容器可以被销毁重建，而容器中产生的数据和文件则可能需要在重建后的容器里继续使用。这种情况比较常见于自研发服务的场景，当服务新版本发布而使得容器更新重建时，有些文件需要保留。其次，有些数据文件需要在多个容器中共享，比较典型的是互为消费者和生产者的服务，通过某个目录中的文件进行数据交换。此外，在容器默认情况下采用的层叠式的联合文件系统虽然提供了容器运行所需的镜像层复用能力，但在读写效率上却比原本的 Ext4 或 Fat32 等普通文件系统下降许多。因此即使有些读写频率比较频繁的内容，比如存放日志、缓存的目录，与其直接写在容器里，不如从外部挂载一个普通文件系统的目录更合适。

Docker 对卷存储的支持主要经历了两个阶段，分界点是 Docker 1.9 版本。在这个版本以前，Docker 中与卷存储相关的特性无非就是 `docker run` 的 `--volume` 和 `--volumes -from` 参数，以及 Dockerfile 的 VOLUME 语句。运用模式也相当单一，挂载数据的来源主要有两

种，要么是一个本地磁盘的目录（用户指定或自动临时创建），要么是另一个容器里的特定目录。“数据容器”是这个阶段的产物，它主要是为了解决给数据目录命名和管理的问题，通过使用一个专门存放数据的容器，再将这个容器里的目录挂载到真正需要数据的容器里，当这个服务被重建或删除后，新的容器可以继续使用数据容器的名字方便地加载之前存储的数据。同时，由于数据容器的指定目录是从主机匿名挂载的，该目录也具有比较高的读写效率。用命令来表示使用数据容器的过程如下所示。

```
$ docker run -dt --volume /data --name demo_data alpine
$ docker run -d --volume-from demo_data --name demo_service demo
```

图 6-11 直观地展示了这种数据挂载方式。

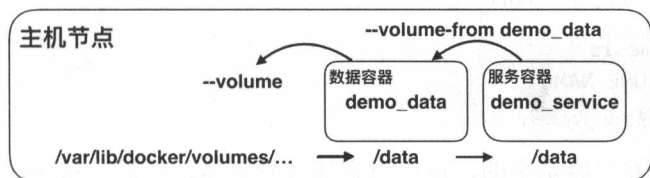


图 6-11 “数据容器” 示意图

Docker 1.9 版本增加了两个与存储卷相关的特性：存储卷管理命令和存储卷驱动插件。

`docker volume` 是 Docker 中用来管理存储卷的命令。它包括一系列的子命令，例如使用 `docker volume ls` 将列出当前节点上所有可见的数据卷，在一个经常运行各种容器的节点上执行该命令通常会打印出一串长长的数据卷列表，如下所示。

```
$ docker volume ls
DRIVER      VOLUME NAME
local       106398a55765543dc78c8...
local       120b75f5cb9529123f1de...
local       27c51dab7e760301aafb5...
```

这是因为数据卷的生命周期通常是长于容器的。在 Docker 早期版本中挂载本地数据卷有两种方式：“指定主机目录和容器目录”或者“仅仅指定容器目录”。这两种方式创建的数据卷实际上都是“匿名”的，用户无法使用名字来在别的地方引用它。对于前一种情况，由于已经指定了挂载点在主机上的位置，容器结束后，主机上相应目录里的内容自然会被保留。而对于后一种情况，当容器结束以后，它写到数据卷里的内容其实也依然存在，只不过它们被藏在了 Docker 安装目录下的一些以数据卷 ID 命名的目录里，想要再拿出来复用就不太容易。现在 Docker 帮助我们把这些数据卷统一管理起来，并且用 ID 作为它们的名字。

有了名字就可以复用数据卷了，只要把这串长长的名字放在原本指定主机目录的位置（`--volume` 参数值的冒号前面），就可以在另一个容器里挂载同一个数据卷，如下所示。

```
$ docker run -it --volume 106398a557655.....85aca57ccc8889b863f:/data alpine
```

在 Docker 1.9 以及之后的版本中，用户可以直接创建具有有意义名称的数据卷。`docker volume create` 是用来创建数据卷的命令，这样实际上就把数据卷的创建和使用分离开了，如下所示。

```
$ docker volume create --name vol1
```

使用 `docker volume ls` 查看数据卷列表后将能够看到这个卷的名字不再是一长串 ID 值，而是用户指定的名称“vol1”。

```
$ docker volume ls
DRIVER      VOLUME NAME
local       vol1
```

新创建的数据卷内容是空的，可以将它挂载到容器，然后向里面写入一些文件，如下所示。

```
$ docker run --rm -v vol1:/data alpine sh -c "echo hello > /data/f.txt"
```

这个容器执行完以后会自动结束并被销毁（使用了 `--rm` 参数），但它所挂载的数据卷并不会随着容器消失而消失。下面这个命令启动了另一个新的容器，并且从刚刚的那个数据卷里读取之前写入的数据。

```
$ docker run --rm -v vol1:/data alpine sh -c "cat /data/f.txt"
hello
```

容器集群本质上是由许多提供容器服务的节点组成的庞大服务体。容器集群的使用者并不会关心实际的容器被启动在集群的哪个节点上，但集群中的节点有时会出现计划的或非预期的停机，此时集群管理工具通常会将运行在这个节点上的所有容器自动迁移到集群里的其他节点上。然而如果有一些容器使用了本地的数据卷，当容器迁移的时候就会发生数据丢失的问题。采用网络存储磁盘或分布式存储磁盘是一种可行的办法。通过将 NFS、Ceph RBD 等设备挂载到节点的特定目录，然后再将这些目录挂载到容器里，能够在一定程度上解决这种单节点失效的尴尬情况。但出现节点故障时，仍然需要经由人工操作，把网络存储设备重新挂载到另一个节点上，这样的方式实际上还是将特定容器限制在了特定的节点。

数据卷驱动插件的出现使得第三方的底层存储能够方便地与 Docker 的数据卷应用深

层关联起来,从而实现网络存储卷自动随容器迁移。社区贡献了包括 GlusterFS、Ceph RBD、NFS、GCE、Azure 等云平台的开源存储驱动。还有一些驱动提供了多种网络存储后端的选择,例如 Convoy (支持 NFS/VFS/EBS)、Contiv (支持 NFS/RBD) 和 Flocker (支持 EBS/GCE/Cinder/……)。其中 ClusterHQ 公司的 Flocker 算得上是最早进入这个领域的产品,2015 年和 2016 年的 DockerCon 大会上都有它的身影。例如在一个部署了 Flocker 服务的集群里,可以这样创建一个网络数据卷,如下所示

```
$ docker volume create --driver flocker --name vol2
```

查看数据卷列表,vol2 卷的驱动类型被标记为了 flocker,这个数据卷在 Flocker 集群中的任意一个节点上都能够被看到,如下所示。

```
$ docker volume ls
DRIVER      VOLUME NAME
local       vol1
flocker     vol2
```

此时可以在集群里重复刚刚的实验,首次创建一个容器,在数据卷里写入一些内容,如下所示。

```
$ docker run --rm -v vol2:/data alpine sh -c "echo hello > /data/f.txt"
```

到另一个节点上,用同样的名字挂载并读取该数据卷的内容,如下所示。

```
$ docker run --rm -v vol2:/data alpine sh -c "cat /data/f.txt"
hello
```

此时,Flocker 驱动会在后台自动地将所需数据卷相应的磁盘先前使用过的节点卸载,挂载到当前节点,然后启动容器,而这一切对于使用者而言都是透明的。

本质上来说,在各种容器存储驱动的背后,集群所使用的分布式存储形式与过去相比并无特别之处,传统的 SAN、NAS 和近年来比较流行的 HDFS 等存储服务都可以作为容器集群的存储实现。HDFS 存储主要用在大数据领域,特别是与 Hadoop 或 Spark 等经典框架结合。虽然通过 Fuse 机制也能将 HDFS 存储作为普通文件系统挂载使用^①,但由于 HDFS 主要为大吞吐量的数据进行设计,对于频繁读写文件的场景性能较差,通常不作为集群文件系统的替代品,更像是一种特殊的对象数据库。

SAN (Storage Area Network, 存储区域网络) 与 NAS (Network Attached Storage, 网络接入存储) 是两类比较常见的通过网络发布存储资源的形式。也许是因为英文缩写相似,

① <https://wiki.apache.org/hadoop/MountableHDFS>

它们常常被放在一起比较，但实质上是两种完全不同网络协议的存储形式。SAN 最初是指使用高速光纤组成的存储设备专用网络，以块设备的形式提供存储能力，传输的是原始二进制 I/O 数据。随着 TCP/IP 协议的普及，出现了基于以太网络的 IP-SAN 和 Server SAN，现在有许多开源的 SAN 方案，例如 GFS、GlusterFS、Ceph 等都不再依赖于光纤网络协议。NAS 最初集成了处理器和磁盘 / 磁盘柜，类似于文件服务器，用于 TCP/IP 网络上，通过文件存取协议读写数据，例如 NFS、SMB、CIFS 等。其中 NFS 和 SMB 是在以太网上应用最广的两种 NAS 文件存取协议，NFS 用于 Unix/Linux 操作系统的服务，SMB 协议用于 Windows 操作系统的文件服务。

这两类存储方案的最大区别在于它们的访问方法，SAN 是以块 (Block) 级的方式操作，而 NAS 是以文件 (File) 级的方式操作。SAN 提供的块存储设备相当于虚拟磁盘，用户拿到设备后，可以根据需要将它格式化成任何一种文件系统 (如 Ext4、NTFS、Fat32 等)。而 NAS 提供的直接是文件系统，用户需要在本地安装对相应文件系统的驱动，挂载后直接使用。图 6-12 简单地描述了这种区别。

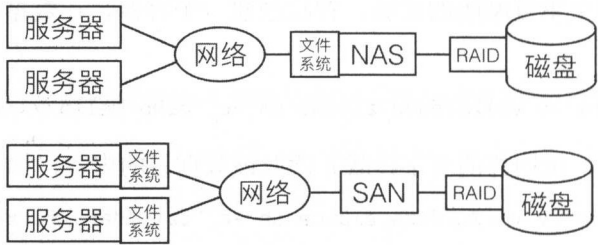


图 6-12 NAS 和 SAN 的结构差异

从使用者的角度来说，更直观的差异在于，由于 NAS 服务的文件系统是存在于远端的，可以实现存储数据的共享访问，即多个容器同时挂载同一个来自网络的存储目录。而 SAN 服务提供的是原始块设备，需在本地进行格式化，每个块设备同时只能同时被一个节点独占，因此同一存储资源无法被多个节点上的容器同时挂载。此外，NAS 服务对于以整个文件为访问单位的场景（如网页、文件传输等）性能较好，而 SAN 则对于需要经常局部更新和修改二进制数据块（如数据库存储、超大日志文件追加写入等）的场景更有效率。

6.2.3 容器卷存储标准

与容器镜像存储的 OCI 规范相比，容器卷存储相关标准的发展相对缓慢。Docker 的卷存储驱动插件接口算是最早的容器卷存储标准，但与 CNM 网络标准一样，它是为 Docker

容器制定的，耦合性强，不具有通用性。

直到 2017 年中旬，在 Kubernetes 社区的牵头下，第一个综合多厂商存储模型的开放式容器存储标准——容器存储接口（Container Storage Interface，简称 CSI）诞生了。这个姗姗来迟的业界接口规范，明确规定了容器存储领域的通用术语，卷存储对象的生命周期、部署方式和接口格式，避免了各种容器卷存储方案在部署、使用、管理等方面五花八门的差异。与 CNI 标准一样，CSI 标准的内容开源在独立的 Github 仓库中^①。

从整体来看，CSI 标准将存储插件的服务（依据其实现的接口）分为两类，分别是用于与编排系统主节点交互的 Controller Plugin 和用于完成实际存储分配任务的 Node Plugin。其中 Controller Plugin 是可选的，如果存储插件没有需要额外集中管理的元数据，可以仅实现 Node Plugin 服务。存储插件和编排系统之间采用 gRPC 二进制协议通信。

存储卷的挂载过程被划分为十分简洁的三个阶段，如下所示。

- CREATED：存储卷准备完毕，等待挂载。
- NODE_READY：需要挂载卷的服务准备完毕。
- PUBLISHED：挂载完成。

这种粗略的划分反映出目前 CSI 标准还处于相对宽松的时期。值得一提的是，虽然在集群中，容器卷存储的主要挑战点在于容器被调度系统迁移后的数据重新挂载，但 CSI 标准并非仅仅是仅仅针对跨节点分布式存储设计的，它同样适用于使用本地数据卷存储的场景。Kubernetes 的 CSI 文档页面收集了一些主流存储方案的 CSI 插件，以及可供参考的开源插件示例^②。

截至本书完稿时，CSI 规范的内容刚刚发布 v 0.1.0 版本。作为首个采用 CSI 标准的容器编排系统，Kubernetes 从 1.9 版本开始以 Alpha 特性形式提供 CSI 支持。考虑到该标准的发展尚需一段时间，本书第 6.2.4 至第 6.2.6 小节的示例部分将使用较成熟的 Docker 卷存储驱动插件来介绍容器存储的运用场景。

6.2.4 基于 NFS 的卷存储

NFS 是网络文件系统（Network File System）的简称，属于一种比较简单的 NAS 协议。它的第一个版本由 Sun 公司开发并开源，目的是让不同操作系统的计算机可以共享数据，是目前 Linux 下最常见的一种网络存储协议，最新的协议版本是 Version 4。

^① <https://github.com/container-storage-interface/spec/blob/master/spec.md>

^② <https://kubernetes-csi.github.io/docs/Drivers.html>

使用 NFS 并不一定需要专业的存储服务器，在普通的 Linux 发行版上都可以安装 NFS 服务，然后将本地磁盘的存储资源以 NFS 协议暴露到集群里，供其他节点接入使用。主流 Linux 发行版的包管理器（例如 Apt 或 Yum）里都内置了 NFS 的驱动和后台服务，直接使用相应命令安装即可。此外，也可以通过 Docker 容器部署 NFS 服务，如下所示。

```
$ docker run --privileged -d --name=nfs \
  -p 111:111/tcp -p 111:111/udp \
  -p 2049:2049/tcp -p 2049:2049/udp \
  -v /data:/nfs-share flin/nfs4-alpine
```

在这个容器中，需要将容器的 2019 和 111 端口映射到主机，它们分别被 NFS 主服务和用于管理 NFS 子功能端口的 Portmap 服务使用，客户端通过这两个服务的配合才能进行完整的数据通信。

在集群的另一个节点上可以使用 mount 命令挂载这个远程文件系统。首先，检查系统是否安装了 NFS 的客户端驱动，如果系统中存在“/sbin/mount.nfs”这个文件，则说明驱动已经就绪。否则应该通过系统相应的包管理器进行安装。例如 Ubuntu 系统所需的包是 nfs-common，如下所示。

```
$ ls /sbin/mount.nfs
ls: /sbin/mount.nfs: No such file or directory
$ sudo apt-get install nfs-common
... ..
$ ls /sbin/mount.nfs
/sbin/mount.nfs
```

然后，执行挂载操作。下面这个例子将来自“10.1.2.3”节点上通过 NFS 服务暴露的文件系统的根目录挂载到当前节点的“/mnt/nfs”位置。

```
$ sudo mkdir /mnt/nfs
$ sudo mount -v -t nfs -o vers=4 10.1.2.3:/ /mnt/nfs
```

挂载后查看“/mnt/nfs”目录中的内容，应与远端节点共享的目录内容一致。若是在多个节点上同时挂载同一个远端节点共享的同一个文件系统，在其中任意一个节点上修改挂载目录的内容，都会同步体现在其他节点上，即所有挂载该文件系统的目录内容将保持一致。

真实的 NAS 服务设备结构要比上面所演示的复杂得多，不过只要使用的是 NFS 协议，从客户端看来，所有的网络都可以使用相同的方法挂载使用。

不难想到，如果手动挂载存储集群的 NFS 服务到计算集群的节点上，然后将挂载的目录映射到创建的容器，便可实现容器跨节点漂移时的数据重用。只不过这种操作会涉及多个过程的协作，在实际使用时必须小心翼翼地处理各个主机挂载点的位置和容器的关系，

稍不小心就容易发生目录冲突和数据串线等麻烦的事情。

各种主流的集群解决方案都各自提供了不同的适配网络存储的措施。为了避免陷入某种特定解决方案的讨论，下面仅仅介绍如何在 Docker 原生的环境下使用网络的卷存储。Docker 对各种不同的卷存储进行了抽象，并允许第三方以驱动插件的方式封装存储的实现细节，仅暴露存储接入能力。虽然这种特性在 Docker 中早已实现，但由于 Docker 默认只内置有一个名为“local”的卷存储驱动插件，平时挂载卷时从来不会显示所用的插件类型，也几乎感觉不到这些插件的存在。

在 Docker 文档里有一个长长的卷存储驱动的插件列表^①，由于容器的本地存储已经没有什么文章可做，这些插件主要都实现了基于网络的存储接入，其中有不少直接支持 NFS 协议的存储资源。下面以其中通用性较好的 Netshare 插件为例，演示一下插件是如何简化容器的网络卷挂载过程的。

Netshare 使用 Go 语言编写，部署和使用都比较简单。只需从它的 GitHub 发布页面下载最新版本二进制文件到系统 PATH 变量中的任意一个目录里，然后赋予可执行权限即可，如下所示。

```
$ sudo wget -O /usr/local/bin/docker-volume-netshare \
  https://github.com/ContainX/docker-volume-netshare/releases/download/v0.34/
  docker-volume-netshare_0.34_linux_amd64-bin
$ sudo chmod +x /usr/local/bin/docker-volume-netshare
```

与 Docker 的网络插件一样，卷存储插件同样需要一个常驻的进程，Netshare 命令支持很多类型的网络存储协议，对于 NFS 的存储，应使用 `docker-volume-netshare nfs` 命令。直接执行这个命令时，它会在前台运行，一旦控制台关闭服务就自动结束了，因此建议将它注册为系统服务保持在后台运行。下面以采用 Systemd 管理服务的系统为例，创建相应的 Service 文件，然后启动服务即可。

```
$ cat <<EOF | sudo tee /etc/systemd/system/netshare-nfs.service
[Unit]
Description=Netshare NFS Service
After=docker.service
[Service]
ExecStart=/usr/local/bin/docker-volume-netshare nfs
[Install]
WantedBy=multi-user.target
EOF
$ sudo systemctl start netshare-nfs
```

① https://docs.docker.com/engine/extend/legacy_plugins/#volume-plugins

一切准备就绪，现在可以创建要挂载数据卷的容器了。挂载使用的参数依然是 `-v` 或 `--volume`，只不过在冒号前面的挂载数据源格式变成了“<NFS 服务地址>/<NFS 服务目录>”，其中“NFS 服务目录”部分若为空则挂载 NFS 服务器共享的根目录。真正的不同之处在于，还需要一个额外的参数 `--volume-driver`，这个参数会告诉 Docker，挂载数据卷的时候要使用名称是“nfs”的插件提供的存储功能，这样 Netshare 就能接收到用户的挂载请求，自动完成将存储资源挂载到主机然后映射到容器的过程，如下所示。

```
$ docker run -it --rm --volume-driver=nfs \
-v 10.1.2.3/dir1:/mount \
alpine /bin/sh
```

在这个容器的“/mount”目录里的操作都会同步到“10.1.2.3”这个 NFS 服务共享目录下的“/dir1”子目录里，如下所示。此时如果观察节点上挂载的磁盘，也会发现 Netshare 其实自动在主机上挂载了 NFS 服务器的“dir1”目录，不过现在不再需要自己维护容器和主机挂载位置的关系了，就像是容器直接挂载了远程的数据卷一样。

```
$ sudo mount | grep dir1
10.1.2.3:/:/dir1 on /var/lib/docker-volumes/netshare/nfs/10.1.2.3/dir1 type nfs4
(rw,relatime,vers=4.0,... ...)
```

6.2.5 基于 Ceph 的卷存储

使用 NFS 服务共享存储十分方便，特别当提供 NFS 服务的是专用的 NAS 集群时，还可以结合 RAID 等功能实现磁盘阵列的无单点故障，或是在容量不够时插入更多磁盘进行动态扩容。不过，基于 NAS 的网络存储协议，不论是 NFS 还是 Samba，由于文件系统是在服务器端构建的，客户端（比如使用存储的容器）在挂载存储时会自动获得对存储服务磁盘分区所有空间的使用权利。若是很多容器共用一个 NAS 服务，即使各自挂载的是不同的目录，相互看不到对方的文件，但只要有一个容器无节制地写入数据，将 NAS 存储服务共享的磁盘空间耗尽，所有使用这个 NAS 服务的容器就都要遭殃。磁盘限制容量的方式通常有分区、配额（Quota）等，都要对文件系统进行操作，而 NAS 存储的文件系统不被客户端控制，使此类问题变得十分麻烦。

相比之下，基于 SAN 的存储方案将文件系统的创建和管理交给了客户端处理，因此能够很好地解决磁盘容量分配的问题。此外，SAN 暴露的是原始的 I/O 块读写能力，客户端可以根据实际使用场景，将存储资源格式化成恰当的文件系统格式，在数据存取效率上也更具有优势。

Ceph^①是一种开源的通用分布式存储系统，它最初是加州大学圣克鲁兹分校的 Sage Weil 博士在 PhD 期间的研究项目，目的是实现一种能够扩展到 PB 量级，同时保持高性能和高可靠性的存储方案。Ceph 的底层使用了基于对象的存储机制，它被称为 RADOS (Reliable, Autonomic, Distributed Object Store, 可靠的自动化分布式对象存储)，这种存储结构是 Ceph 设计的最大亮点所在，它确保了整体系统具有极高效的水平扩展能力，并提供了对大量数据设备的存储能力抽象。Ceph 在 RADOS 层之上提供了一组名为 LibRADOS 的原始 API，这组 API 可以用来对 RADOS 对象进行十分细致的操作。LibRADOS 的 API 虽然可以直接存取数据，但它与平时使用的文件系统和存储系统都差别太大，因此，在这层 API 之上，Ceph 还提供了三种形成的高层 API，分别以对象数据库 (RADOS Gateway)、块设备 (Reliable Block Device) 和文件系统 (Ceph File System) 的形式提供具体的存储服务。这种分层提供存储能力的结构如图 6-13 所示。

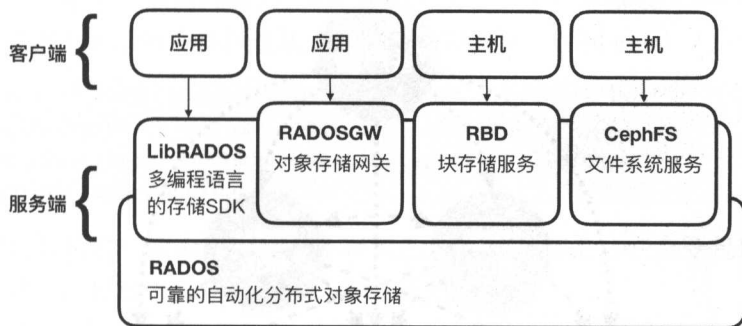


图 6-13 Ceph 的存储和 API 结构

从部署的角度上看，用户不论是通过对象存储、块存储还是文件存储形式接入 Ceph 系统，都需要一个用于指令交互和数据传输的程序（称为 Ceph 的“客户端 (Client)”），其中 Ceph 块存储的客户端已经被纳入 Linux 内核（从 Linux 2.6.34 版本开始），这也是现在 Ceph 被广泛接受的一个原因。在服务端方面，Ceph 的结构略为复杂，它主要由以下五类服务进程组成。

- 监控器 (Ceph Monitor, MON): 用于监控、维持和提供整个存储集群的结构和状态，以及存储位置的映射计算参数。这种映射只在集群结构发生改变的时候才会变化，因此客户端通常只需获取一次映射参数，之后就可以直接从其他服务节点获取数据，从而实现分流。
- 管理器 (Ceph Manager, MGR): 从监控器中独立出来的新服务，用于将整个集群

① <http://ceph.com/>

项目最初出现在他的一篇博客^①里。由于 Sebastien 是 Ceph 项目的核心贡献者之一，并且在红帽公司负责另一个 Ceph 自动化部署的项目 Ceph-Ansible，这篇博客发表后引起了许多人的兴趣，Ceph-Docker 项目也被 Ceph 收入到官方 GitHub 仓库下^②。经过几年的发展，Ceph-Docker 已经被许多用户接受，成为一种常见的 Ceph 部署方式。

在 Docker Hub 上可以找到 Ceph-Docker 项目的官方预构建镜像，不过由于没有采用自动构建发布，它的版本有时会落后于 GitHub 较多。建议直接从 GitHub 仓库进行构建，如下所示。

```
$ git clone https://github.com/ceph/ceph-docker.git
$ docker build -t ceph/daemon ceph-docker/ceph-releases/luminous/ubuntu/16.04/daemon
```

这个镜像包含了 Ceph 所需要的所有服务的可执行程序以及相关的命令行工具。有些命令行工具在挂载和操作 Ceph 存储时会用到，可以通过带 `-i -rm` 参数启动的容器来直接获得这些命令，并把命令所需的一些目录从主机挂载到容器里，例如 `ceph -s` 这个命令相当于：

```
$ docker run -i --rm --privileged --pid host --network host \
--volume /dev:/dev --volume /etc/ceph:/etc/ceph \
--volume /sys:/sys --volume /var/lib/ceph:/var/lib/ceph \
--entrypoint ceph ceph/daemon -s
```

每次执行这么长的一串命令显然是不可取的，不过只要把这些命令包装成与原本命令同名的脚本，就可以轻松地使用它们了，如下所示。

```
for file in ceph rbd rados radosgw-admin ceph-conf; do
    echo | sudo tee /usr/bin/$file >/dev/null <<EOF
    #!/bin/sh
    docker run -i --rm --privileged --pid host --network host \
    --volume /dev:/dev --volume /etc/ceph:/etc/ceph \
    --volume /sys:/sys --volume /var/lib/ceph:/var/lib/ceph \
    --entrypoint \$(basename $0) ceph/daemon "$@"
EOF
    sudo chmod +x /usr/bin/$file
done
```

基本的命令准备就绪后，创建一个用来存储 Ceph 配置的 Etcd 服务，Ceph 对 Etcd 的支持是在 Luminous（即 v12.0.0）版本以后加入的，它解决了此前创建 Ceph 集群的时候，需要在各个节点间拷贝配置文件以及同步配置信息的麻烦。简单起见，这里通过容器启动一

① <http://www.sebastien-han.fr/blog/2013/09/19/how-I-barely-got-my-first-ceph-mon-running-in-docker/>

② <https://github.com/ceph/ceph-docker>

个单节点 Etcd 服务，如下所示。

```
$ docker run -d --name etcd --network host flin/etcd:v3.1.8
```

同样地，可以用这个镜像模拟一个 Etcd 操作命令 `etcdctl` 的可执行文件，如下所示。

```
echo | sudo tee /usr/bin/etcdctl >/dev/null <<EOF
#!/bin/sh
docker run -i --rm --network=host \
    flin/etcd:v3.1.8 etcdctl "$@"
EOF
sudo chmod +x /usr/bin/etcdctl
```

对于提供块存储服务的 Ceph 集群，至少需要三种服务：MON、OSD 和 MGR。默认情况下，OSD 的每份数据要保存三份副本（相当于每个数据备份两份），因此提供 OSD 服务的节点至少需要三个，其余的 MON 和 MGR 服务至少各一个即可。其中 MON 服务管理了存储集群的块映射表信息，因此在配置客户端时会用到它的 IP 地址。下面假设运行 MON 的节点是 172.31.30.100，运行 Etcd 服务的节点是 172.31.30.200，当然它们其实可以运行在同一个节点上，两个服务之间没有冲突。

首先在任意一个节点上，执行一次容器中的 `populate_kvstore` 命令，它会在 Etcd 服务里添加最基本的配置信息，如下所示。

```
$ docker run --rm --network host \
    -e KV_TYPE=etcd \
    -e KV_IP=172.31.30.200 \
    -e KV_PORT=2379 \
    ceph/daemon populate_kvstore
```

这些配置保存在 Etcd 的 “/ceph-config/ceph” 路径下面，如下所示。

```
$ etcdctl --peers 172.31.30.200:2379 ls -r /ceph-config/ceph
/ceph-config/ceph/global
/ceph-config/ceph/global/max_open_files
/ceph-config/ceph/osd
/ceph-config/ceph/osd/osd_journal_size
/ceph-config/ceph/auth
/ceph-config/ceph/auth/cephx
```

三个作为 OSD 服务的节点（如果作为测试，可以复用 MON 或 Etcd 服务的节点，它们之间没有排斥关系）需要至少挂载除根路径磁盘以外的一块磁盘，因为根路径所使用的磁盘需要被系统本身使用，是不可以被格式化掉再作为存储资源对外提供的。假设使用节点的 “/dev/xvde” 磁盘作为共享存储，使用容器的 `zap_device` 命令，将它的文件系统信

息删除，如下所示。

```
$ docker run --rm --privileged \
-v /dev/./dev/ \
-e OSD_DEVICE=/dev/xvde \
ceph/daemon zap_device
```

这个步骤是可以选的，因为在 OSD 服务启动后还会对块设备进行清空。

最先启动的应该 MON 服务，它会向 Etcd 中创建更多的配置属性，并且在系统的“/etc/ceph”目录下生成内容相同的一些配置文件。有了这些配置以后，才可以创建其他的

```
$ docker run -d --name mon \
--network host \
-v /var/lib/ceph:/var/lib/ceph \
-v /etc/ceph:/etc/ceph \
-e MON_IP=172.31.30.100 \
-e CEPH_PUBLIC_NETWORK=172.31.0.0/16 \
-e KV_TYPE=etcd \
-e KV_IP=172.31.30.200 \
-e KV_PORT=2379 \
ceph/daemon mon
```

然后在三个存储节点上分别启动运行 OSD 的容器，如下所示。

```
$ docker run -d --name osd \
--network host \
--privileged \
--pid host \
-v /dev/./dev/ \
-e OSD_DEVICE=/dev/xvde \
-e OSD_TYPE=disk \
-e OSD_FORCE_ZAP=1 \
-e KV_TYPE=etcd \
-e KV_IP=172.31.30.200 \
-e KV_PORT=2379 \
ceph/daemon osd
```

最后在任意一个节点上启动一个 MGR 服务，如下所示。

```
$ docker run -d --net=host \
--name mgr \
-e KV_TYPE=etcd \
-e KV_IP=172.31.30.200 \
-e KV_PORT=2379 \
```



```
ceph/daemon mgr
```

这样一个简易的 Ceph 集群就完成了，可以在运行 MON 服务的节点上使用 `ceph -s` 命令查看集群的运行状态，如下所示。

```
$ ceph -s
cluster 81a08a20-9e55-4223-97f4-45727188fa21
health HEALTH_OK
monmap e4: 3 mons at {...}
election epoch 8, quorum 0,1,2 ...
mgr active: ...
osdmap e11: 3 osds: 3 up, 3 in
pgmap v18: 64 pgs, 1 pools, 0 bytes data, 0 objects
322 MB used, 485854 MB / 486176 MB avail
64 active+clean
```

如果要在其他节点上使用 `ceph` 命令，除了这些节点上要有该命令（可以用容器里执行），还应将 MON 节点上生成在 `/etc/ceph` 目录里的所有配置拷贝到其他节点里，如下所示。

```
$ sudo scp -r root@172.31.30.100:/etc/ceph /etc/
```

现在在任意一个有 Ceph 配置文件的节点上，都可以创建并挂载 Ceph 远程提供的块设备存了。在 Ceph 存储中还有一些专用的概念，比如存储池（pool）是一种逻辑上的存储资源组，数据卷（image）是在存储池中创建的实际可分配存储块，Ceph 使用了英文中的单词“Image”而不是更常见的“Volume”来表示卷，在使用的时候需要注意上下文以便和容器的“镜像”加以区分。映射（map）指的是将远端存储资源变成本地块设备的操作，此外还有快照（snapshot）和从快照创建卷（clone）等操作。

存储池是 Ceph 集群逻辑上的隔离单位，不同的存储池可以有不同的存储参数，如副本数（Replication Size）、PG（Placement Groups）参数、CRUSH 参数等，并且拥有独立的数据卷和快照资源。在部署完 Ceph 后，会自动创建一个名称是“rbd”的默认存储池，可以用 `rados lspools` 命令查看，如下所示。

```
$ rados lspools
rbd
```

使用 `rbd create` 命令来创建服务器上的存储卷并指定大小，如下所示。注意，虽然这个存储卷的划分依然在服务端执行，但是它的容量大小和其他参数都是由客户端发出的命令指定的，因此客户端实际上具有对存储卷和文件系统的控制权。

```
$ rbd create demo_image --size 1024 --image-feature layering
```

这里的 `--image-feature` 参数实际上是为了减少一些 Ceph 默认的存储特性，只保留

最基本的 layering 特性，否则在之后进行 map 操作时，可能会在有些环境里遇到由于内核不支持所需特性而发出的“RBD image feature set mismatch”错误。

使用 `rbd ls` 命令能列出当前集群中所有的数据卷，如下所示。

```
$ rbd ls
demo_image
```

查看这个数据卷的详细信息，如下所示。

```
$ rbd --image demo_image info
rbd image 'demo_image':
  size 1024 MB in 256 objects
  order 22 (4096 kB objects)
  block_name_prefix: rbd_data.102374b0dc51
  format: 2
  features: layering
  flags:
```

在正式挂载 Ceph 块设备前，还需要执行 `modprobe` 命令加载内核中的 `rbid` 驱动模块，如下所示，否则在执行 `rbid map` 时会出错，这个内核模块就是图 6-14 中提到的客户端。

```
$ sudo modprobe rbd
```

使用 `rbid map` 命令将服务端的数据卷映射成本地设备，如下所示。

```
$ rbd map demo_image
/dev/rbd0

$ rbd showmapped
id pool image      snap device
0  rbd  demo_image -  /dev/rbd0
```

此时得到的还是一个裸的块设备，在使用之前，需要给它加一个文件系统，比如 Linux 下比较常用的 Ext4 文件系统格式，如下所示。

```
$ sudo mkfs.ext4 /dev/rbd0
mke2fs 1.42.13
Discarding device blocks: done
... ..
Allocating group tables: done
Writing inode tables: done
Creating journal (8192 blocks): done
Writing superblocks and filesystem accounting information: done
```

现在就可以像挂载本地设备一样直接挂载这个块设备了，如下所示。

```
$ sudo mkdir /mnt/ceph
```

```
$ sudo mount /dev/rbd0 /mnt/ceph
```

在挂载后的目录里随意创建一些文件和内容，然后将这个磁盘卸载，如下所示。

```
$ echo 'hello ceph' | sudo tee /mnt/ceph/hello
$ sudo umount /dev/rbd0
```

接着在另一个磁盘重新映射并挂载同一个服务器上的数据卷，会发现之前写入的内容全部被保留了下来。这样就实现了数据的跨节点漂移，如下所示。

```
$ rbd map demo_image
$ sudo mkdir /mnt/ceph
$ sudo mount /dev/rbd0 /mnt/ceph
$ cat /mnt/ceph/hello
hello ceph
```

那么如果在前一个节点没有卸载之前，就在另一个节点上又挂载了这个 Ceph 数据卷，会发生什么呢？许多人的直觉是，要么挂载失败，要么像 NAS 那样在挂载后两边内容保持同步。但实际情况是，第二个节点是能够挂载成功的，在后一个节点在挂载时，能够看到前一个节点在挂载时刻之前做的所有改动。但在这以后，任意一个节点上对这个挂载点的修改都只同步到服务器，但不会再同步到其他的节点上，两个节点看到的内容将是不一致的。当两边的数据卷都卸载后，服务端存储的实际上是两个节点各自修改的叠加，如果两个节点同时修改了同一个文件，则后修改的那个会最终留在服务器端的存储里。这与块设备本身的特性有关系，所有基于 SAN 原理的块存储都有类似特点，要么无法同时多处挂载，要么挂后不同步，因此在使用的时候需要比 NAS 设备更加小心。有兴趣的读者可以尝试一下，这里就不详细给出操作命令了。

如果在刚刚的例子中，将从 Ceph 存储服务器挂载到本地的磁盘再挂载到容器里面作为数据卷，不也就实现了容器漂移后数据跟着移动了吗？从原理上来说没有问题，只是这里同样涉及复杂的卸载、挂载、映射和目录管理的麻烦事。在目前的各种主流容器集群方案里同样集成了诸如 Ceph、GlusterFS 等分布式块存储的支持，与 NFS 类似，下面抛开这些特定集群的分布式存储能力，只以原生 Docker 方式谈一谈如何简化这种数据漂移过程。

在 Docker 刚推出存储插件功能时，社区里出现过好几款开源的 Ceph 存储插件，但大多出于个人开发者之手，已经不再更新了。目前开源的存储插件中，对 Ceph 支持较好的还有思科 (Cisco) 公司推出的 Contiv^①和戴尔 (Dell) 公司“EMC{code}”团队维护的 RexRay^②。

① <https://github.com/contiv/volplugin>

② <https://github.com/codedellemc/rexray>

前者的部署略复杂，且仅支持 Redhat 系列操作系统，相对而言 RexRay 的通用性更好一些，下面简单介绍它的用法。

Docker 插件总要有个后台服务，RexRay 也不例外。从 RexRay 的网站上可以下载它在各种主流 Linux 版的二进制包，更简单的方式是用官方的这个脚本部署，如下所示。

```
$ curl -sSL https://dl.bintray.com/emccode/rexray/install | sh -
```

这个服务需要在每一个使用分布式存储的节点上部署，然后为它创建一个配置文件，EMC 提供了一个在线的配置生成工具 REX-Ray Configuration Generator^①，如图 6-15 所示，将生成的配置内容放到节点的“/etc/rexray/config.yml”文件里。

Create REX-Ray Configurations with Ease

Enter your information and watch the configuration be generated automatically

Step 1: Download REX-Ray

```
$ curl -sSL https://dl.bintray.com/emccode/rexray/install | sh
```

Step 2: Create the configuration

REX-Ray Logging Level: Warn

Add A Service: Ceph RADOS Block Devices [Add]

Ceph RBD

Pool Name: rbd

Default Pool: rbd

```
libstorage:
  service: rbd
  rbd:
    defaultPool: rbd
```

hint: hover over labels to get more information about individual options

download copy

Step 3: Place the file at

```
/etc/rexray/config.yml
```

Step 4: Run as a service

```
$ rexray start
```

图 6-15 RexRay 的在线配置生成工具

由于 RexRay 还不支持像 Etcd 这样的统一配置存储，因此不得不把生成的配置文件在每个节点上分别拷贝一份，然后在各个节点启动 RexRay 服务，如下所示。

```
$ sudo rexray service start
```

这个命令会根据当前的系统自动调用相应的服务管理器启动 RexRay 服务，例如在使用 Systemd 管理系统服务的 Linux 发行版上，它相当于 `sudo systemctl start rexray`。

然后就可以在 Docker 里直接创建 `rexray` 类型的存储卷了，如下所示。

```
$ docker volume create --driver=rexray --name=demo-rbd --opt="size=5"
```

① <http://rexrayconfig.emccode.com/>

末尾的`--opt` 参数中的内容是传递给 RexRay 后台服务的，`size=5` 的单位是 GB，也就是说生成一个大小为 5GB 的存储卷。这个卷实际上是创建在 Ceph 存储集群上的，因此在每个运行了 RexRay 服务的节点上都能看到它，如下所示。

```
$ docker volume ls
DRIVER      VOLUME NAME
rexray      demo-rbd
... ..
```

运行一个数据库服务，并将持久化存储的目录挂载到 Ceph 上，如下所示。

```
$ docker run -d -v demo-rbd:/var/lib/mysql mysql
```

6.2.6 使用公有云存储

随着越来越多企业将基础设施全面云化，除了一部分对数据安全和保密性要求较高的行业，许多企业不再自建存储集群，转而使用更稳定、廉价的公有云存储资源。因此包括 Swarm、Kubernetes、Mesos、Rancher 等容器集群方案都或多或少地内置了对例如 AWS、GCE、Azure、Digital Ocean 等主流云平台的存储集成，许多开源的容器存储插件也都提供了相应的功能。相对而言，国内的公有云在与容器周边生态的对接方面做得还不太好。

目前在存储服务方面做得最全的云平台是 AWS，它提供了三种通用的存储服务，如下所示。

- EBS：全称是弹性块存储服务（Elastic Block Store），提供按磁盘类型和使用存储用量计费的即接即用的块设备存储功能。用户可以根据对数据读写压力的需求选择不同种类的磁盘。
- EFS：全称是弹性文件系统（Elastic File System），提供基于 NFS 协议的文件存储，用户可以将其视为存储量近乎无限大的海量存储池，只需按照实际的存储用量付费，但它的价格是这三者中最高的。
- S3：全称是简单对象存储服务（Simple Storage Service），它的每 GB 存储价格在三者中相对最低，不过需要按实际存储用量以及读写的流量两个维度计费。使用 S3 服务作为文件存储时，实际上是使用一种基于 S3 的文件系统协议 S3FS，以类似于 NFS 的方式挂载到系统中。

下面继续以原生 Docker 和 RexRay 插件举例。RexRay 存储插件可以同时支持这三种存储服务，不过每个 RexRay 后台服务只能同时提供其中一种存储类型，因此如果想混用这

几种网络存储，就得部署好几个 **RexRay**，在前一小节中介绍的部署方式就不太合适了。为了解决这个问题，**RexRay** 设计了各种存储类型服务的容器镜像，并且按照标准 **Docker** 扩展插件的方式^①发布到了 **Docker Hub** 上。

在 **AWS** 的 **IAM**（Identity and Access Management，认证和访问管理中心）页面创建一对 **API** 访问的密钥，并授予 **EBS**、**EFS** 和 **S3** 的访问权限。然后就可以用 **docker plugin install** 命令来启动相应的服务容器了，如下所示。

```
$ docker plugin install rexray/ebs \
  EBS_ACCESSKEY=<AWS 的访问密钥> \
  EBS_SECRETKEY=<AWS 的保密密钥> \
  EBS_REGION=<存储所在的 AWS 区域>

$ docker plugin install rexray/efs \
  EFS_ACCESSKEY=<AWS 的访问密钥> \
  EFS_SECRETKEY=<AWS 的保密密钥> \
  EFS_REGION=<存储所在的 AWS 区域> \
  EFS_SECURITYGROUPS=<AWS 区域相应的安全组> \
  EFS_TAG=rexray

$ docker plugin install rexray/s3fs \
  S3FS_ACCESSKEY=<AWS 的访问密钥> \
  S3FS_SECRETKEY=<AWS 的保密密钥> \
  S3FS_REGION=<存储所在的 AWS 区域>

$ docker plugin ls
```

ID	NAME	DESCRIPTION	ENABLED
c96c8747c5c3	rexray/ebs:latest	...	true
ea9140848046	rexray/efs:latest	...	true
704921eab595	rexray/s3fs:latest	...	true

注意，**AWS** 的 **EFS** 服务目前只在少数的几个区域开放，如果插件启动后发现运行不正常，请检查当前使用的区域是否在官方文档^②中已经支持。

然后使用不同的驱动类型分别创建存储卷，如下所示。

```
$ docker volume create --driver rexray/ebs --name demo-ebs
$ docker volume create --driver rexray/efs --name demo-efs
$ docker volume create --driver rexray/s3fs --name demo-s3fs
$ docker volume ls
```

① <https://docs.docker.com/engine/extend/>

② <https://aws.amazon.com/efs/pricing/>

DRIVER	VOLUME NAME
rexray/ebs:latest	demo-ebs
rexray/efs:latest	demo-efs
rexray/s3fs:latest	demo-s3fs
...	...

创建完存储卷后，这些卷的信息实际上都是保存在 AWS 服务上的。此外若在不同的节点上使用相同的命令创建 RexRay 存储服务，也可以看到这些已创建的卷，因此这些卷中存储的数据都是跨节点共享的。创建一个容器，将这几个存储卷分别挂载到不同目录，就会看到这几种类型的数据卷虽然从用户的角度上看挂载方式几乎相同，其底层挂载原理是有很大的差别的，如下所示。

```
$ docker run --rm -it \  
-v demo-ebs:/data-ebs \  
-v demo-efs:/data-efs \  
-v demo-s3fs:/data-s3fs \  
ubuntu bash  
# mount | grep "/data"  
/dev/xvda1 on /data-s3fs type ext4 (...)  
/dev/xvdf on /data-ebs type ext4 (...)  
172.31.33.183:/data on /data-efs type nfs4 (...)
```

在这次的整个过程中，我们没有自己搭建分布式存储服务，完全通过云端资源实现了多节点共享的容器卷存储。在实际应用时，应当根据具体情况选择适当的存储服务实现以及存储的本地接入方式。

6.3 本章小结

本章详细讲解了容器集群中的两个十分重要领域：网络和存储。并通过一些开源组件的例子，演示了跨节点的容器网络和存储的使用。

由于网络延时和空间距离等客观存在的因素，如何进一步优化在分布式环境中的网络和存储的应用效率、稳定性依然是一个非常值得持续探讨的问题。这是近年来许多企业级的容器平台解决方案不断发力的一个方向，并且依然有很大的改进空间。可以预见在这两个技术领域里，未来还将诞生许多推动容器以及虚拟化发展创意和产品。

第 7 章 容器服务的基础设施

除了存储和网络，为了支撑容器集群的正常运作，还需要一些必备的基础设施。这一章将涉及其中的四个主要方面，包括监控告警、日志管理、服务发现和镜像管理，它们分别在容器集群生态中做到了各自对立又相辅相成的功能职责。

容器化的服务集群与传统服务集群在运维方式上存在不少差异。容器对服务的运行时隔离使得每个服务拥有独立的运行上下文、网络栈和文件存储空间，几乎就是一个微型虚拟机。使用者在创建容器实例时通常会为每个容器起一个直观且有意义的名字，这些信息对于排查问题时定位出错的服务十分有用。因此，针对容器集群设计基础设施服务所需要考虑的因素势必与传统服务集群有所不同。本章将针对这些问题，介绍当下比较流行的开源解决方案。

值得说明的是，为了不与特定的集群方案绑定，这一章将尽量让所有的例子仅使用 Docker 基本的容器化能力来部署这些集群辅助服务，以确保每个小节里介绍的工具和方案都是能够独立使用的。这些例子只需稍加适配即可以使用在 Swarm、Kubernetes、Mesos 或 Rancher 的集群里。尤其是在一些局部服务间相互通信的地方，示例会将容器的端口映射到主机，并使用到主机的 IP 地址，这种做法在真正的集群场景里是应该尽量避免的。主流的容器集群解决方案都内置有 DNS 模块，提供自动将部署的服务名转换为实际容器 IP 地址的功能，只要合理使用是完全可以避免使用端口映射和固定 IP 地址的。

7.1 集群性能监控

7.1.1 常见的开源性能监控方案

说到监控的开源方案，最先想到的就是经典的 Zabbix 或者 Nagios。那么对于容器的监控来说哪一种更好呢？如果你已经对其中某一款十分熟悉，好消息是它们都有插件可以用

作容器集群的监控，不过功能上只能算勉强足够，而且社区对容器支持方面的改进似乎并不太积极。如果你对这两款软件都不太熟悉，没有关系，接下来会推荐两款更加适合用于容器集群性能监控和事件告警的开源方案：在 2015 年的 Docker 欧洲技术大会（Docker EU 2015）上，Swisscom AG 的云方案架构师 Brian Christner 推荐过的 TICK Stack（当时只有 InfluxDB）和 Prometheus。

TICK Stack^①是 Influxdata 公司推出的集数据采集、存储、图表可视化和指标告警于一体的端到端方案，包括 Telegraf、InfluxDB、Chronograf、Kapacitor 四个子项目。目前 TICK Stack 分成商业版和开源版，虽然它的官网已然充满浓浓的商业气息，但 Influxdata 承诺这套工具栈将保持开源。

既然要讲 Prometheus^②，就顺便先说说 CNCF^③组织。CNCF 组织的全称是云原生计算基金会（Cloud Native Computing Foundation），它在 2015 年 7 月成立，由 Google 牵头，最初有包括 IBM、Intel、Cisco、VMware、Docker、Mesosphere 等在内的 19 家企业组成，目前已经扩展到近百位云计算相关企业。它的设立宗旨是促进整个云原生应用的发展和标准化，并为支持分布式、可扩展的应用所需的各种组件及其组装方式制定规范，以及给这个蓝图提供一组参考的基础设施架构。作为一个软件基金会，CNCF 不定期地从社区中找到并接纳与之目标一致的项目进行投资。CNCF 已经接纳了例如由 Google 贡献的容器集群项目 Kubernetes、由 Docker 公司贡献的容器虚拟化实现项目 Containerd 等重量级成员。作为监控领域的新星，Prometheus 也成为了最早一批入选 CNCF 的开源项目。与 TICK Stack 类似，Prometheus 同样提供包括数据采集（Exporter）、数据存储、展示（PromDash）和告警（AlertManager）的完整监控解决方案。

从监控的方式来说，容器监控的实施方式可分为在主机上收集容器数据和在容器内部收集数据两种。在容器刚开始流行时，由于很多企业实际将容器当作虚拟机管理，在容器内部署额外监控组件的做法屡见不鲜，这样能够最小化用户对容器的学习和感知成本。但在容器里采集数据的方式对容器镜像构建有很强的侵入性，且使得每个容器中同时运行多个进程，本身是一种典型的反模式。从实际的效果来看，大量的数据收集客户端也会增加很多额外的系统开销。随着容器概念的普及以及容器性能信息获取逐渐容易起来（例如 `docker stats` 命令和相关接口），目前需要在每个容器内采集性能数据的情况已经很少见了。取而代之的是以主机节点为单位，让每个采集装置（Agent 或 Exporter）汇总整个节点

① <https://www.influxdata.com/products/open-source/>

② <https://prometheus.io>

③ <https://www.cncf.io>

中的所有容器信息，这样做的另一个好处是统一了容器与主机数据采集的过程，由于通常对硬件资源的约束实际是发生在主机层面上的，若在汇报容器数据时能将相应的节点数据进行关联，在发生告警和奔溃时将有助于更全面地提供用于调查问题的信息。

7.1.2 基于 TICK Stack 的性能监控

TICK Stack 中的四个组件（图 7-1）实际是独立存在的，它们之间使用标准的 API 交互。现在可以只用其中一部分，或者将其中的某个组件替换成其他的兼容方案，从而最大限度地利用企业现有基础设施。

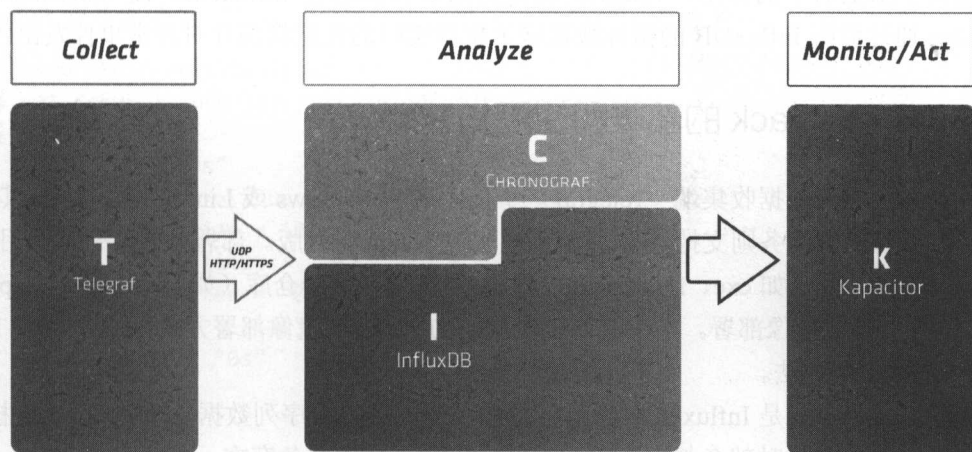


图 7-1 TICK Stack 的四个组件

Telegraf 项目采用插件机制提供了通用的性能数据采集服务。通过内置的模块，它支持几十种常用软件的性能数据采集，甚至能够直接兼容 Prometheus 的 Exporter 数据格式，从而直接将任何支持 Prometheus 的数据源变成自己的数据源。在数据的汇聚方面，Telegraf 也并非仅仅支持存储数据到 InfluxDB，而是提供了包括 Graphite、Graphite、Kafka、MQTT 等许多开放式的存储目标。

InfluxDB 项目是 TICK Stack 中最早的产品，在其他几个项目还未诞生前，InfluxDB 通过集成社区开源产品 cAdvisor 和 Grafana 组成完整的容器监控技术栈。作为一种专用的时间序列数据库，InfluxDB 具有极佳的序列写入和查询性能，且能够自动清理一段时间以前的过期数据。它提供与 SQL 相似的查询语法，可以写出基于时间段的、十分灵活而复杂的查询条件。

Chronograf 项目是一个前端基于 React.js 编写的性能数据可视化服务，不仅能够方便用户查询和展示指标数据曲线，还提供了丰富的模板和工具让用户快速设置好许多常用服务的监控看板。此外，Chronograf 还是用户创建告警规则和事件处理脚本的好帮手，实用的在线编辑器可以避免用户手动在配置文件里编写复杂的条件语句和判断逻辑。

Kapacitor 项目提供了告警触发和处理的功能。它内置了包括邮件、HipChat、Slack、OpsGenie 等多种通知、响应服务的支持，能够帮助用户快速构建一套应急事务处理机制，以便在第一时间自动化地解决线上事故，并在无法修复时及时告知相关人员。

这四个组件的关系可以通过图 7-1 表示出来，它们都是使用 Go 语言编写的，且在部署、配置方式上都十分相似。从数据流向看，作为提供数据写入和查询的组件，InfluxDB 在结构上起到了承上启下的作用，Telegraf 将数据采集后推送进 InfluxDB，而 Chronograf 和 Kapacitor 通过监听 InfluxDB 的指标数据展示集群实时的性能状态并对异常事件发出告警。

7.1.3 TICK Stack 的部署和使用

TICK Stack 的数据收集端 (Telegraf) 可以运行在 Windows 或 Linux 系统上，而其他的存储、展示、告警服务则支持 Mac 系统和各主流 Linux 发行版。部署的方式包括使用在相应环境上的安装包（如 exe、rpm、deb 文件）、通过包管理器仓库（如 brew、yum、apt 等）以及使用 Docker 镜像部署。本小节以比较通用的 Docker 镜像部署方式为例，介绍 TICK Stack 的基本使用方法。

首先需要部署的是 InfluxDB 服务，它存储了监控的时间序列数据，是整个系统的核心，其他几个组件在启动时都会与它连接，因此应该先于其他服务存在。在启动容器前，得创建它的配置文件，为了简化例子，尽可能使用官方镜像，在节点上创建这个文件再挂载到容器中，如下所示。

```
$ sudo mkdir /etc/influxdb
$ cat <<EOF | sudo tee /etc/influxdb/influxdb.conf
[meta]
  dir = "/var/lib/influxdb/meta"
[data]
  dir = "/var/lib/influxdb/data"
  wal-dir = "/var/lib/influxdb/wal"
EOF
```

注意这种简易配置的 Influxdb 是没有包含用户和密码对访问进行验证的，在实际使用的场景中，强烈推荐增加身份验证的设置^①。在通过容器部署 Influxdb 时，存储数据的

^① https://docs.influxdata.com/influxdb/v1.2/query_language/authentication_and_authorization/

“/var/lib/influxdb”目录也应该挂载到主机，如下所示。

```
$ docker run -d --name influxdb \
  -p 8086:8086 \
  -v /etc/influxdb/influxdb.conf:/etc/influxdb/influxdb.conf \
  -v /var/lib/influxdb:/var/lib/influxdb \
  influxdb:1.2
```

然后在需要采集数据的节点上部署 Telegraf 服务。Telegraf 的采集配置比较复杂，分成许多独立的区域，每个区域都有相应的配置项，完整的配置有几千项之多（在它的 GitHub 仓库中有一个参考配置文件，包含了十分详细的注解说明）。假设监控的内容只是当前节点的 CPU 和磁盘使用情况以及当前节点上运行的容器的性能统计信息，且用于存储的 InfluxDB 服务地址为 172.31.13.238。以下是一个极简的配置参考。

```
$ sudo mkdir /etc/telegraf
$ cat <<EOF | sudo tee /etc/telegraf/telegraf.conf
[agent]
  interval = "10s"
  round_interval = true
  metric_batch_size = 1000
  metric_buffer_limit = 10000
  collection_jitter = "0s"
  flush_interval = "10s"
  flush_jitter = "0s"
  debug = false
  quiet = false
  omit_hostname = false
[[outputs.influxdb]]
  urls = ["http://172.31.13.238:8086"]
  database = "telegraf"
  write_consistency = "any"
  timeout = "5s"
[[inputs.cpu]]
  percpu = true
  totalcpu = true
  collect_cpu_time = false
[[inputs.disk]]
  ignore_fs = ["tmpfs", "devtmpfs"]
[[inputs.docker]]
  endpoint = "unix:///var/run/docker.sock"
  container_names = []
  timeout = "5s"
  perdevice = true
```

```
total = false
EOF
```

启动 Telegraf 服务，如下所示。

```
$ docker run -d --name telegraf \
  --network host \
  -v /etc/telegraf/telegraf.conf:/etc/telegraf/telegraf.conf \
  telegraf:1.3
```

Chronograf 和 Kapacitor 的部署顺序可以随意。Chronograf 容器的 Web UI 默认使用 8888 端口，如下所示。

```
$ docker run -d --name chronograf \
  -p 8888:8888 \
  -v /var/lib/chronograf:/var/lib/chronograf \
  chronograf:1.3 --influxdb-url=http://172.31.13.238:8086
```

启动 Kapacitor 服务的方法类似，同样建议将它的数据存储目录挂载出来，如下所示。

```
$ docker run -d --name kapacitor \
  -p 9092:9092 \
  -v /var/lib/kapacitor:/var/lib/kapacitor \
  kapacitor:1.3
```

通常来说，在实际的环境里，Telegraf 应该运行在所有需要采集数据的节点，既包括运行用户任务的集群节点，也包括 TICK Stack 自己运行的节点。其他三个服务只需要部署一份，考虑到 InfluxDB 节点的磁盘读写比较频繁，建议单独部署一个节点（或几个节点，不过开源版的 InfluxDB 只提供了有限的高可用支持，详见项目在 GitHub 的文档^①），而 Chronograf 和 Kapacitor 在为了节省资源的情况下，可以共用节点。

在部署了 Influxdb 服务的节点上进入 Influxdb 容器中的命令行，用 `influx` 命令连接到数据库控制台，其中的 `-precision rfc3339` 参数表示在查询其中记录的序列数据时使用 RFC3339 格式表示时间，如下所示。

```
$ docker exec -it influxdb influx -precision rfc3339
Connected to http://localhost:8086 version 1.2.2
InfluxDB shell version: 1.2.2
>
```

进入控制台后，就可以对 Influxdb 进行操作了，例如，列出所有可用的数据库，如下所示。

^① <https://github.com/influxdata/influxdb-relay/blob/master/README.md>

```
> SHOW DATABASES
name: databases
name
----
telegraf
_internal
```

使用 `USE` 切换当前使用的数据库，如下所示。

```
> USE telegraf
Using database telegraf
```

Influxdb 数据库中的数据是按照“序列”来组织的，例如，查看在 `telegraf` 数据中包含的序列，如下所示。

```
> SHOW SERIES
key
---
cpu,cpu=cpu-total,host=node1
... ..
```

在数据库中查询和筛选数据的语法与 SQL 非常相似，但不需要使用分号结尾，如下所示。

```
> SELECT "host","usage_idle" FROM "cpu" WHERE "usage_idle" < 20 LIMIT 10
name: cpu
time                host      usage_idle
----                -
20XX-XX-XXTXX:XX:XXZ node1    17.5463194792199
... ..
```

此外，Influxdb 也提供了基于 HTTP 的 API 方便其他客户端调用，如下所示。

```
$ curl -G 'http://localhost:8086/query?pretty=true' --data-urlencode "db=telegraf"
--data-urlencode 'q=SELECT usage_idle FROM cpu LIMIT 1'
{
  "results": [
    {
      "statement_id": 0,
      "series": [
        {
          "name": "cpu",
          "columns": [
            "time",
            "usage_idle"
          ],
          "values": [
```

```
[
  "2017-05-02T12:44:50Z",
  99.80000000000018
]
}
```

通过浏览器打开 Chronograf 服务所在节点地址的 8888 端口。单击左侧的“Host List”按钮，进入所有被监控节点列表的页面，随便单击进入一个节点。不需要额外的配置，Chronograf 就能自动展示出从该节点上采集到的各主要指标数据的时间序列图表，如图 7-2 所示。在左侧的“Dashboard”菜单中，用户可以创建自定义的图表面板，将任意查询结果进行组合并实时展示和更新。

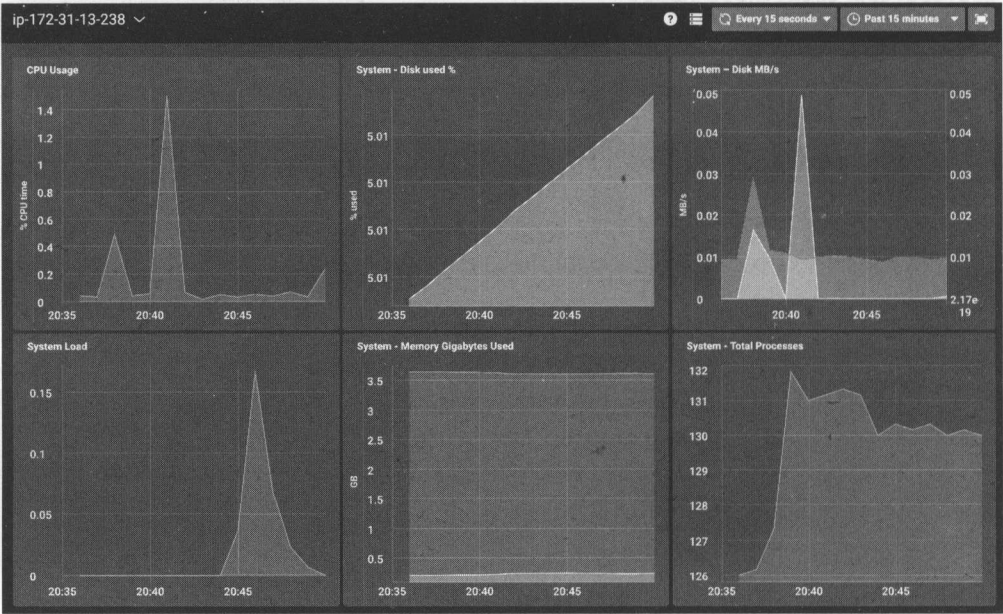


图 7-2 Chronograf 的数据展示页

TICK Stack 的告警功能是通过 Kapacitor 服务实现的，但有了 Chronograf 的面板后，用户不再需要手动修改 Kapacitor 配置。单击左侧的“Alerting”菜单，首次进入时会提示需要关联的 Kapacitor 服务，单击“Add Kapacitor”按钮，填写 Kapacitor 服务地址和各种告

警通知方式的参数，保存后就可以在相应的子菜单里可视化地添加告警规则并查看告警记录了，如图 7-3 所示。

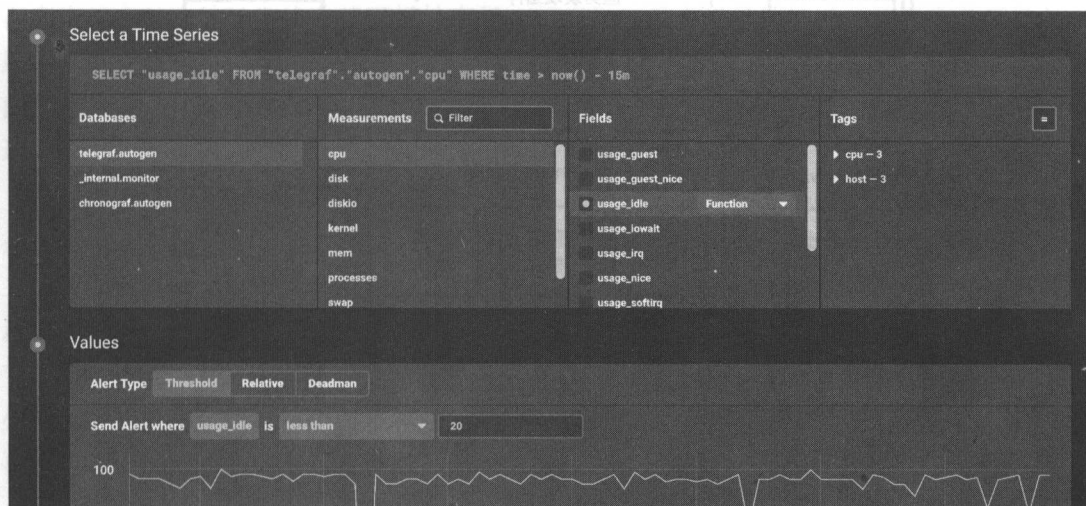


图 7-3 在 Chronograf 中配置告警规则

整体来说，TICK Stack 这套监控工具自身包含端到端的服务组件，因此各部分的部署和使用方式都比较统一，在易用性方面也做得比较到位。

7.1.4 基于 Prometheus 的性能监控

相比 TICK Stack，在社区里 Prometheus 的知名度要大得多。一方面是因为 Prometheus 采用了完全开源的策略，没有什么遮遮掩掩的东西，对社区用户提出的各种需求反应速度很快。另一方面则是因为 Prometheus 技术成熟得较早，积累了许多一线用户，其中不乏像 DigitalOcean 和 CoreOS 这样的明星企业，使得后来者在使用时更加放心，也许 CNCF 的光环也起到了一些推波助澜的作用，或多或少地增加了它的曝光度。

出于这个现状，本小节也将使用比 TICK Stack 更多的笔墨来介绍 Prometheus 的部署和使用。

图 7-4 展示了 Prometheus 的基本结构。在这个图中，处于中间位置的是 Prometheus Server，它在整个架构中也处于绝对的核心地位。相比 TICK Stack 中各组件独立管理各自配置的模式，在 Prometheus 体系中，几乎所有的配置都是在 Prometheus Server 端完成的。这样做的好处是，用户能够一站式地修改整个集群中与监控有关的各种配置参数，随着监控节点数目的增加，一旦要进行涉及大量节点采集参数的配置变更，该模式的优势就会体

现得十分明显。

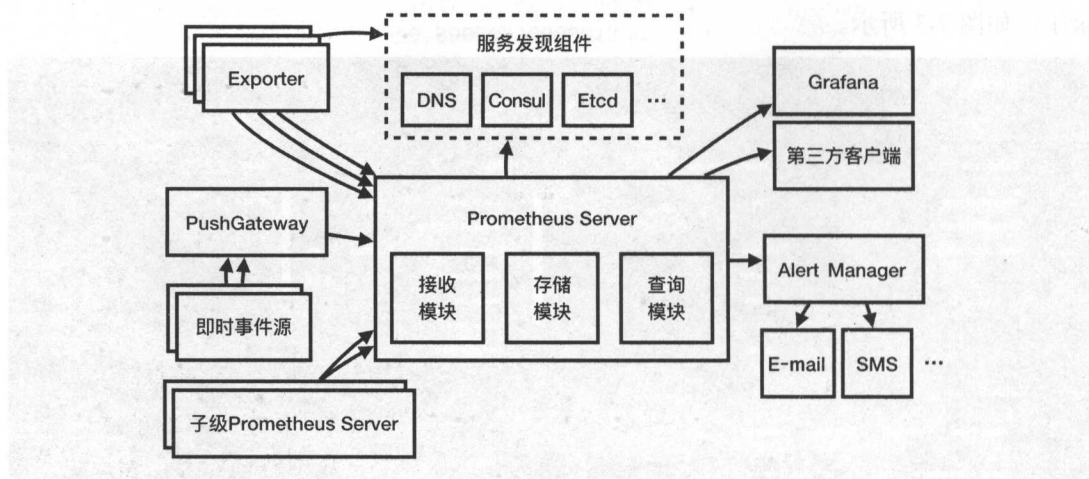


图 7-4 Prometheus 的部署结构

具体来说，TICK Stack 体系是由每个节点的 Kapacitor 服务分别配置自己的采集指标，然后将采集到的数据推送给 InfluxDB 存储，Kapacitor 服务自己配置告警规则并对数据的异常变化进行监控和处理。而在 Prometheus 体系里，各个节点部署的采集器只是在收到外界请求时，才将指标数据采集上来响应给请求方，而何时采集、采样间隔、采集哪些数据都是由数据的获取端（也就是 Prometheus Server 服务）统一管理的。不难看出，这是一种由中央节点向所有受控节点“拉取”数据的模式，与传统监控自下而上“推送”数据的模式有很大不同。此外，Prometheus 告警的规则和监控数据异常的行为同样在中心的 Prometheus Server 服务中进行配置和触发，而这个体系中的告警处理模块仅仅负责将已经发生的事件信息对接到各种处理单元或第三方系统中。

图 7-4 的左侧部分是数据的采集端。数据的来源主要有几类，分别是 Exporter、PushGateway 以及子级的 Prometheus Server 节点。

Exporter 就是普通的数据采集器，根据采集的数据类型不同，分成很多种类。与 Telegraf 的单个服务适配所有数据来源的方式不同，每种 Exporter 都是一个独立运行的服务进程，并且只采集特定类型的数据，官方提供的 Exporter 包括监控节点基本性能数据的 Node Exporter、监控网络服务性能的 Apache Exporter、Nginx Exporter、HAProxy Exporter 等，以及监控数据库运行性能的 MySQL Exporter、PostgreSQL Exporter、RethinkDB Exporter、MongoDB Exporter 等几十种采集器^①。

^① <https://prometheus.io/docs/instrumenting/exporters/>

PushGateway 是一类特殊的数据采集器，它用于解决某些 Prometheus 的拉取模型不适用的场景。例如某些偶发性的数据（比如短暂的流量异常）若采用拉取的采集方法，往往在查询状态时这种异常状态就已经结束了，因而导致丢失记录的情况。或者有些受监控的服务只能在事件发生时推送数据到指定目标，而无法进行事后查询的情况。PushGateway 能够接收并缓存任意数据源主动推送上来的数据，并提供一个事后查询的接口，从而将“推送”模型的采集数据器转换成“拉取”模型的服务。

子级 Prometheus Server 节点是 Prometheus 级联结构的产物，每个 Prometheus Server 自身也提供有符合 Exporter 数据协议的接口，这种设计主要是为了解决规模化的问题。由于主动拉取数据的采集方式使得中心节点（Prometheus Server 服务）的资源开销相比传统推送方式更高，因而单个 Prometheus Server 服务能够同时处理的节点数目比较有限（与每个节点的数据量、网络环境等有关，通常一个中心服务器只能处理几百个节点的数据）。若要处理更大的集群，可以将集群依据特定规则分别汇聚到多个不同中心节点上，然后在上层再从这些子级的中心节点对数据进行二次汇聚。

在图 7-4 上方的服务发现（Service Discovery）组件提供了采集端服务注册的功能。这种机制是为了解决每次添加或删除采集端时，都需要手动修改相应 Prometheus Server 服务采集配置的问题。当有新的采集端启动时，只需将自己的信息添加到服务发现组件，并定期向它发送保活心跳包，当有采集端被移除时，要么它可以主动删掉自己在服务发现中的记录，要么相应的记录会由于长时间收不到心跳包而被删除。Prometheus Server 服务则监听服务发现组件中的数据，一旦有内容变化，就自动更新采集的目标范围，从而实现监控策略与监控目标的关联解耦。

图 7-4 右侧的内容主要是监控数据的可视化展示和基于规则的告警处理组件。在数据可视化方面，Prometheus 最初使用的是专用的管理界面 PromDash，后来由于这个基于 Ruby on Rails 的项目自身的运行性能太差，已经停止维护，官方推荐的替代产品是独立的开源数据面板 Grafana。与告警处理相关的组件是 Alert Manager，不过该组件不负责告警规则的设定和告警的触发（这部分在中间的 Prometheus Server），仅仅提供了将发生的告警事件与相应处理程序、邮件、聊天软件以及其他第三方消息平台对接的服务。

7.1.5 Prometheus 的部署

Prometheus 服务的各个组件同样主要通过 Go 语言来编写，它的大多数组件（除了个别与系统相关的数据采集组件，比如 Node Exporter）都可以部署在 Windows、Mac 和各种 Linux 发行版上。不过，Prometheus 直接提供的是二进制文件打包下载，没有各发行版定制

的安装包，因此不能通过系统的包管理器进行安装。不过由于都是用 Go 语言编写的文件，所有服务的启动都是拆包即用的。相比之下，使用 Docker 部署 Prometheus 则有不少需要注意的技巧问题。考虑到和容器集群配合的使用场景，本小节将主要介绍通过 Docker 来部署 Prometheus 的方式。

在上个小节中提到过 Prometheus Server 是整个 Prometheus 监控体系的配置和管理中心，按理说应该首先启动它。但在当前的设计中，Prometheus Server 和 Alert Manager 服务之间的关联是通过 Prometheus Server 的启动参数来指定的（这个设计略不合理，若能使用配置文件指定会方便得多），为了避免在创建 Alert Manager 后返工修改 Prometheus Server 的启动参数，可以先创建 Alert Manager。

在使用 Docker 创建服务时，通常将与数据、配置和日志相关的目录挂载到主机。Alert Manager 的日志是直接打印到标准输出的，不再需要单独处理，这里将主机的“/var/lib/alertmanager”和“/etc/alertmanager”目录作为存放 Alert Manager 数据和配置的位置，如下所示。建议将这两个目录的所有者修改为非 root 用户。

```
$ sudo mkdir -p /var/lib/alertmanager /etc/alertmanager/template
$ sudo chown -R $(whoami):docker /var/lib/alertmanager /etc/alertmanager
```

由于把配置文件的目录挂载到了本地的一个空目录，在启动 Alert Manager 之前，还需要在这个目录里准备一个基础配置文件。Alert Manager 的配置分为全局属性（global）、通知模板（templates）、告警路由规则（route）、告警抑制规则（inhibit_rules）、告警接收器（receivers）等部分。在 Alert Manager 的 GitHub 项目 README^①中有一个配置文件的例子，将它稍加修改，保留一条完整的告警规则，如下所示。

```
$ cat <<EOF >/etc/alertmanager/config.yml
global:
  resolve_timeout: 5m
templates:
- '/etc/alertmanager/template/*.tmpl'
route:
  group_by: ['alertname', 'cluster', 'service']
  group_wait: 30s
  group_interval: 1m
  repeat_interval: 1m
  receiver: WebHook
inhibit_rules:
- source_match:
```

① <https://github.com/prometheus/alertmanager>

```

severity: 'critical'
target_match:
  severity: 'warning'
  equal: ['alertname', 'cluster', 'service']
receivers:
- name: 'WebHook'
  webhook_configs:
  - url: 'http://localhost:5001'
EOF

```

这个规则配置的含义是，每隔 30s 将所有从 Prometheus 产生的告警依据名称、发出告警的集群和服务名分组，如果已经发生了“critical”级别的告警，就屏蔽所有相同类型的“warning”级别告警，最后所有的告警都将被发送到另一个 WebHook 服务中进行处理。关于 Alert Manager 配置的更多细节请参考其文档^①。

然后在这个节点上通过 Docker 启动一个 Alert Manager 实例，如下所示。

```

$ docker run -d -p 9093:9093 --name alertmanager \
-v /var/lib/alertmanager:/var/lib/alertmanager \
-v /etc/alertmanager:/etc/alertmanager \
prom/alertmanager:v0.9.1 \
-config.file=/etc/alertmanager/config.yml \
-storage.path=/var/lib/alertmanager

```

在刚刚的告警处理策略里，所有告警在通过相应的聚合和抑制规则后，都被转交给了在 5001 端口监听的另一个服务来处理，稍后再来实现这个服务。这种方式为用户处理告警提供了最大的灵活性，在实际使用时，可以使用发邮件、通知 Slack 或其他 Alert Manager 的接收器作为告警处理的策略。

访问 Alert Manager 所在节点的 9093 端口时会看到一个能够查看所有已触发的告警列表和对特定告警进行抑制的管理页面，如图 7-5 所示。

接下来部署 Prometheus Server。同样地，先在主机上创建用于映射到容器里的数据和配置目录，并转移目录所有者为非 root 用户，如下所示。

```

$ sudo mkdir -p /var/lib/prometheus \
/etc/prometheus/console_libraries \
/etc/prometheus/consoles \
/etc/prometheus/rules
$ sudo chown -R $(whoami):docker /var/lib/prometheus \
/etc/prometheus

```

① <https://prometheus.io/docs/alerting/configuration/>

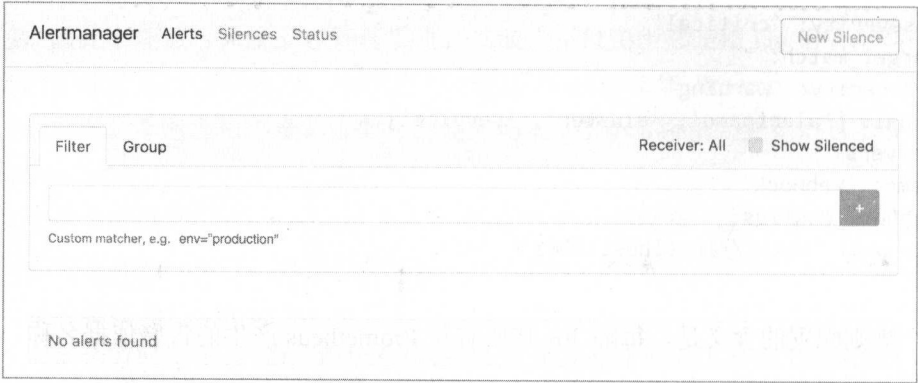


图 7-5 Alert Manager 的管理页面

然后在空的配置文件目录里添加 Prometheus Server 所需的配置文件，如下所示。

```
$ cat <<EOF >/etc/prometheus/prometheus.yml
global:
  scrape_interval: 15s
  evaluation_interval: 15s
  scrape_timeout: 10s
rule_files:
  - '/etc/prometheus/rules/record.rules'
  - '/etc/prometheus/rules/alert.rules'
scrape_configs:
  - job_name: 'prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:9090']
  - job_name: 'nodes'
    consul_sd_configs:
      - server: 'consule-node:8500'
    relabel_configs:
      - source_labels: [__meta_consul_service]
        target_label: job
      - source_labels: [__meta_consul_tags]
        regex: .*,node-[a-z]*,.*
        action: keep
EOF
```

这是 Prometheus Server 的主要配置文件，它的内容与 Alert Manager 配置一样分成几个部分，包括全局配置（global）、数据记录和告警规则文件（rule_files）、采集规则（scrape_configs）等部分。在这个例子中，分别指定了一个额外的数据记录规则和一个告警规则的文件位置，有关这两个文件的内容将在下一小节介绍 Prometheus 使用时再详细说明，现在可以先将它们创建为空文件作为占位。配置文件的末尾定义了两条采集规则。第一条规则采集指定的静态地址，即 Prometheus Server 本身的地址，这样可以实现 Prometheus

Server 的自监控。另一条规则指定将 Consul 作为服务发现来自动集成更多的受监控节点，并进行了一次记录标签的转换，将指标相应的服务名称记录到 `job` 标签，并保留在 Consul 中为服务额外添加的标签信息。

通过 Docker 启动 Prometheus Server 服务，如下所示。

```
$ docker run -d -p 9090:9090 --name prometheus \
-v /var/lib/prometheus:/var/lib/prometheus \
-v /etc/prometheus:/etc/prometheus \
prom/prometheus:v1.8.0 \
-storage.local.path=/var/lib/prometheus \
-config.file=/etc/prometheus/prometheus.yml \
-web.console.libraries=/etc/prometheus/console_libraries \
-web.console.templates=/etc/prometheus/consoles \
-alertmanager.url=http://10.71.31.164:9093
```

注意末尾的 `-alertmanager.url` 参数，它将 Prometheus Server 和 Alert Manager 关联在了一起。浏览 Prometheus Server 节点的 9090 端口就会看到 Prometheus Server 的数据展示界面，提供了告警情况查询、指标查询和绘图、集群采集目标列表和状态查看等功能，如图 7-6 所示。



图 7-6 Prometheus Server 的管理界面

值得注意的是，这个简单的界面只提供信息的查看功能，不能在线修改配置内容，而且查询的结果是不能持久化保存的，页面一刷新就消失。查询持久化的功能需要通过 Grafana 这类面板组件来实现，而配置只能在配置文件里直接修改。

在 Prometheus 的体系里，所有节点的采集配置都是在 Prometheus Server 里管理的，为了能动态地感知集群里新加入的节点和因故障撤出的节点，通常不会把所有节点地址直接写进 Prometheus Server 的配置文件，而是借助服务发现工具来实现动态的节点信息获取。比较常用的服务发现机制有基于 Consul 的、基于 DNS 的和基于特定云平台 API 的等，这里以 Consul 为例。

准备 Consul 的数据存储目录，并通过 Docker 启动，如下所示。

```
$ sudo mkdir -p /var/lib/consul
$ docker run -d -p 8300:8300 \
    -p 8301:8301 \
    -p 8302:8302 \
    -p 8400:8400 \
    -p 8500:8500 \
    -p 8600:8600 \
    -v /var/lib/consul:/consul/data \
    --name=dev-consul \
    consul:0.9.3 agent -dev -client=0.0.0.0 -bind=0.0.0.0
```

在第 7.3 一节介绍服务发现时，将对 Consul 进行更详细的介绍。

这些服务都就绪后，即可在各个集群节点上部署 Exporter 了。Prometheus 社区里存在的各类 Exporter 总数已经有上百种，其中最基本的一些比如节点指标采集、容器指标采集、数据库指标采集等被广泛地使用在不同技术栈的项目中。

Node Exporter 是用于采集服务器节点指标数据的服务，它需要在每一个服务节点上启动。由于它采集的许多数据需要和服务器的硬件资源打交道，因此如果使用 Docker 来启动 Node Exporter，需要正确配置相关的参数，如下所示。

```
$ docker run -d --name node-exporter \
    -v /proc:/host/proc \
    -v /sys:/host/sys \
    -v /:/rootfs \
    --network host \
    prom/node-exporter:v0.15.0 \
    -collector.procfs /host/proc \
    -collector.sysfs /host/sys \
    -collector.filesystem.ignored-mount-points \
    "^/(sys|proc|dev|host|etc)($|/)"
```

相比之下, 如果使用二进制文件直接部署在主机的话, 可以不需要任何额外参数, 直接启动即可。此外, Node Exporter 服务有一个比较有意思的参数是 `-collector.textfile.directory`, 这个参数可以指定一个存放外部指标文件的目录, Node Exporter 会将这个目录下的所有文本文件内容作为指标发送给 Prometheus。

访问部署了 Node Exporter 服务的节点的 9100 端口后会看到一系列的指标数据, 这就是 Prometheus 获取数据的地方。

在所有的集群节点上启动 Node Exporter, 此时 Prometheus Server 并不会自动开始采集这些节点的性能指标。在之前展示过的 Prometheus Server 配置中, 使用了 Consul 作为集群节点信息的来源, 因此还需要将这些节点都注册到 Consul 中, 如下所示。

```
$ curl -X PUT -d '{"ID": "node-1", "Name": "node", "Address": "172.31.31.164", \
  "Port": 9100, "Tags": ["node-db"]}' -i \
  http://consule-node:8500/v1/agent/service/register
$ curl -X PUT -d '{"ID": "node-2", "Name": "node", "Address": "172.31.31.158", \
  "Port": 9100, "Tags": ["node-srv"]}' -i \
  http://consule-node:8500/v1/agent/service/register
```

第 7.3.3 小节会专门介绍 Consul API 的详细用法。在实际情况下, 通常会将节点注册到 Consul 的操作配置为启动后自动执行, 这样每当集群中新增节点时, 就会自动添加到监控列表里。具体的实现方法有将执行节点注册的脚本预置到系统镜像里、使用自动化平台创建节点时进行注册等多种机制。

在各个节点上添加完 Node Exporter 的服务以后, 回到 Prometheus Server 的内置数据展示界面, 搜索时就会看到很多以“node”开头的指标, 这些指标的数据就是通过 Node Exporter 采集到的。

除了节点, 下面专门介绍一下容器指标数据的采集, 因为它使用的采集器有点特殊。按照常量, 采集容器的服务应该被命名为“Container-Exporter”或其他相似的名字, 事实上社区中曾经有过这样的一个项目, 但它已经被标记为弃用, 目前推荐使用 Google 的容器数据采集专用服务: cAdvisor。

和 Node Exporter 一样, 通过容器启动 cAdvisor 服务需要一些技巧, 主要是应该挂载系统的关键数据目录到容器中, 如下所示。

```
$ sudo docker run -d --name=cadvisor \
  -p 8080:8080 \
  -v /:/rootfs:ro \
  -v /var/run:/var/run:rw \
  -v /sys:/sys:ro \
  -v /var/lib/docker:/var/lib/docker:ro \
  google/cadvisor:v0.27.1
```

通过容器启动的 cAdvisor 默认采用 8080 端口，直接访问该端口会看到一个提供当前节点和容器性能数据展示的 cAdvisor，如图 7-7 所示。

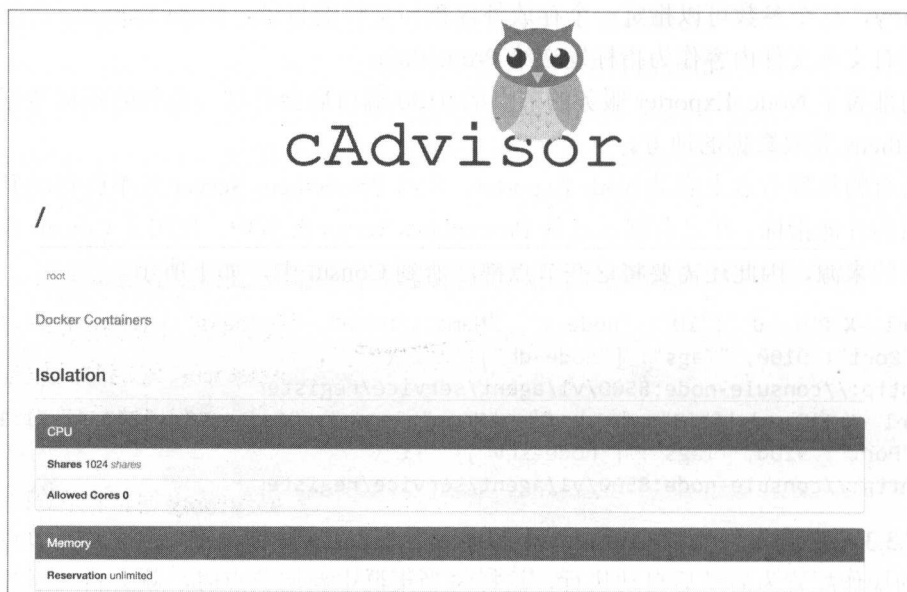


图 7-7 cAdvisor 的单节点性能数据

而访问 cAdvisor 服务 8080 端口的 “/metrics” 路径，则会看到从这个服务上采集到的容器指标数据，如下所示。

```
$ curl -L http://10.78.148.105:8080/metrics
```

容器的监控目标同样需要服务发现的支持，Prometheus 已经内置了 Kubernetes 和 Mesos Marathon 的服务发现机制，同时也可以使用 Consul 这样通用的服务发现支持。将相应配置加到 Prometheus Server 的配置文件 “prometheus.yml” 的末尾。例如同样使用 Consul 作为服务发现的方式，新增一个 “cadvisor” 采集任务，如下所示。

```
- job_name: 'cadvisor'
  consul_sd_configs:
    - server: 'consule-node:8500'
  relabel_configs:
    - source_labels: [__meta_consul_service]
      target_label: job
    - source_labels: [__meta_consul_tags]
      regex: .*,cadvisor,.*
      action: keep
```

然后将容器数据采集器注册到 Consul，方法和 Node Exporter 类似，如下所示。

```
$ curl -X PUT -d '{"ID": "node-1", "Name": "node", "Address": "172.31.31.164", \
"Port": 9100, "Tags": ["node-db"]}' -i \
http://consule-node:8500/v1/agent/service/register
$ curl -X PUT -d '{"ID": "node-2", "Name": "node", "Address": "172.31.31.158", \
"Port": 9100, "Tags": ["node-srv"]}' -i \
http://consule-node:8500/v1/agent/service/register
```

还有一种可以算作 Exporter 的组件，被称为“PushGateway”，它自身不会采集特定的指标数据，而是等待其他服务将数据推送给它，然后将这些数据缓存下来提供给 Prometheus。这个服务的实际作用是将一些发生频率很低的事件性指标（如特定组件是否有网络访问）或是由于历史原因已经存在的推送型的指标采集器，将这些事件源改造成 Exporter 会带来一些不必要的额外成本。对于这些情况，Prometheus 的定时拉取模型就不太适应，因此可以让数据源将指标以事件形式先推送到 PushGateway，然后由 Prometheus Server 从 PushGateway 获取它们。

为了获得更好的性能，同样建议在使用 Docker 部署 PushGateway 时，将它的数据存储目录挂载出来，如下所示。

```
$ sudo mkdir -p /var/lib/pushgateway/storage
$ docker run -d -p 9091:9091 --name pushgateway \
-v /var/lib/pushgateway/storage:/storage \
prom/pushgateway:v0.4.0 \
-persistence.file /storage/file \
-persistence.interval 5m0s
```

PushGateway 监听 9091 端口的“/metrics”路径，接收 POST 请求。通过 curl 命令可以模拟其他数据源将数据推送到 PushGateway 的过程，例如添加一个名称是“pi_metric”、值为 3.14 的单指标 Metrics，如下所示。

```
$ echo "pi_metric 3.14" | curl --data-binary \
@- http://localhost:9091/metrics/jobs/pi_job
```

Prometheus 指标的标签大多是通过在相应指标项上使用{{标签=值}}的语法添加的，除了两个具有特殊含义的标签：job 和 instance。它们是在 Prometheus 从采集器拉取指标时额外附加的。job 标签表示任务的分类，比如通常会将所有的主机指标作为一类任务，所有的数据库指标作为另一类任务等；instance 标签用于标识数据来源，通常是一个 IP 地址或某种具有位置含义的名称。在往 PushGateway 中推送指标时，可以在 URL 上指定这两个指标的值，其中 job 标签是必需的，instance 标签可选，如果没有指定，默认为数据来源的 IP 地址。

现实中的 Metrics 会更复杂，包含多种数据类型和指标注释，比如下面这样的。

```
$ cat <<EOF | curl --data-binary \
  @- http://localhost:9091/metrics/jobs/demo_job/instances/demo_instance
# TYPE demo_metric counter
# HELP demo_metric HELP 开头的注释行会被识别为指标的说明
demo_metric{label="val1"} 42
# 这行是普通注释，下面这个指标自带时间戳
demo_metric{label="val2"} 34 1517414400000
# TYPE another_metric gauge
# HELP another_metric 最常见的普通数值数据就是 gauge 类型的
another_metric 2398.283
EOF
```

注意其中有一个指标“`demo_metric{label="val2"}`”的数据有两列，其中前一列表示的是指标的值，后一列指标时间戳，使用 UNIX 时间戳的格式来表示，精确到毫秒。所以上例中的 1517414400000 转换成秒单位是 1517414400，再转换为公历时间实际上是 2018 年 2 月 1 日。对于没有自带时间戳的指标，Prometheus 存储时将使用数据采集的时间作为指标时间戳。

通过浏览器访问主机的 9091 端口能看到 PushGateway 的数据界面，如图 7-8 所示。

进入 PushGateway 的指标不会被自动移除，但可以手动通过 API 来移除它们。例如，删除采集任务 `demo_job` 中来源为 `demo_instance` 的指标记录，如下所示。

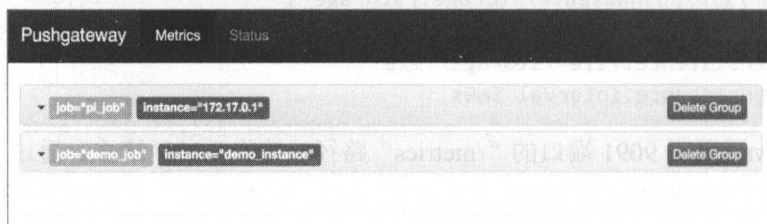


图 7-8 PushGateway 数据展示界面

```
$ curl -X DELETE \
  http://localhost:9091/metrics/jobs/demo_job/instances/demo_instance
```

删除 `demo_job` 采集任务的所有内容，如下所示。

```
$ curl -X DELETE http://localhost:9091/metrics/jobs/demo_job
```

最后要部署的是 Promethues 的展示面板。Prometheus 曾经自己推出过一款使用 Ruby 编写的用户界面：PromDash。后来由于这个界面的响应速度极慢，已经被弃用。现在官方推荐使用社区开发的 Grafana 作为 Prometheus 的标准指标展示面板服务。

Grafana 是一款通用的性能数据展示组件，可以支持 InfluxDB、Prometheus、Graphite

等多种数据源。它的部署十分简单，如下所示。

```
$ docker run -d -p 3000:3000 --name grafana \
-e "GF_SECURITY_ADMIN_PASSWORD=admin" \
grafana/grafana:4.5.2
```

部署完成后访问 Grafana 节点的 3000 端口，使用“admin”用户登录，密码是在启动容器时使用“GF_SECURITY_ADMIN_PASSWORD”环境变量设置的，在前面这个例子里同样是 admin。然后按照界面的引导，依次完成添加数据源、添加 Dashboard、添加用户等操作，如图 7-9 所示。

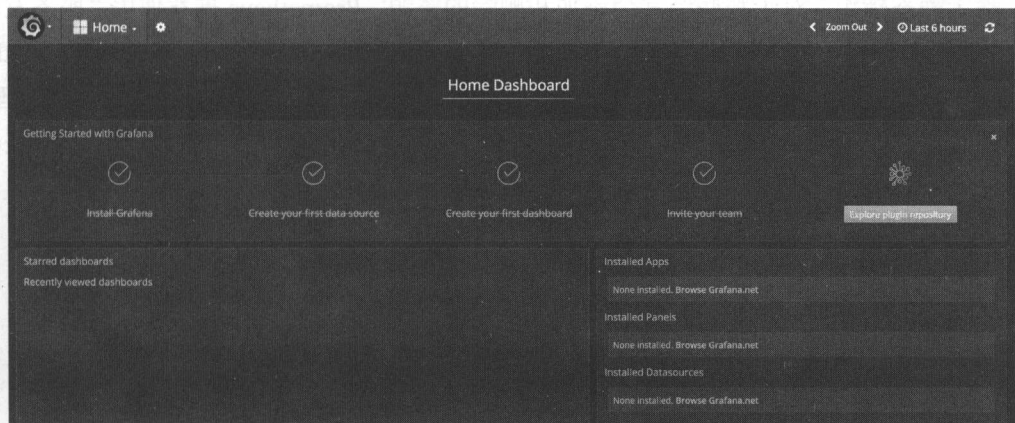


图 7-9 Grafana 的引导页面

7.1.6 Prometheus 的使用

在上一个小节部署 Prometheus 组件时，已经提到过它需要一个配置文件（即“/etc/prometheus/prometheus.yml”），并提供了一个可用的例子。现在来仔细看一下其中的内容。

这个配置文件使用缩进表示配置项的嵌套关系，其中顶级的配置项有三个，如下所示。

- **global** 表示全局的配置属性区，如指定默认的指标获取间隔、采集器访问超时时间等，这些属性大多也可在每个采集任务里单独配置。
- **rule_files** 是额外的规则配置文件的路径，规则文件可以分为“数据记录规则”和“告警触发规则”，它们的作用在稍后进行介绍。
- **scrape_configs** 是与数据采集相关的配置，它是 Prometheus 最核心的功能。

数据采集的配置是使用采集任务（Job）为单元进行管理的，每个任务需要一个名字、指定一种获得采集目标的方式，以及与采集目标和采集内容相关的配置。最简单的获得采

集目标方法是使用静态配置（`static_configs`），下面这个例子中名为“prometheus”的采集任务就使用了这种配置方法。

```
- job_name: 'prometheus'
  scrape_interval: 5s
  static_configs:
    - targets: ['localhost:9090']
```

这个配置实际使用 Prometheus 来监控自己，`targets` 属性的值是所有需要采集的数据源地址。

除了静态指定，对于那些经常会添加和删除的资源，Prometheus 推荐使用“服务发现”的方法来动态更新所管理的目标。Prometheus 支持自动发现和监控 Kubernetes 和 Marathon 等集群平台部署的服务，除此以外也提供了像 DNS 和 Consul 这类通用的服务注册管理方式。下面这个例子中名为“nodes”的任务以及在部署 cAdvisor 后添加的“cadvisor”任务，就是使用 Consul 作为服务源进行配置的（`consul_sd_configs`）。

```
- job_name: 'nodes'
  consul_sd_configs:
    - server: 'consule-node:8500'
  relabel_configs:
    - source_labels: [__meta_consul_service]
      target_label: job
    - source_labels: [__meta_consul_tags]
      regex: .*,node-[a-z]*,.*
      action: keep
```

这部分的配置首先指定了 Consul 服务的地址，然后对采集上来的数据进行了一些标签操作，包括将原始指标中的“__meta_consul_service”标签重命名为“job”标签，然后若“__meta_consul_tags”标签中包含“node-”开头的内容，则保留下来（默认情况下 Prometheus 在存储时会丢弃以两个下划线开头的标签），标签是在检索指标时十分有用的一种辅助工具。

`prometheus.yml` 这个配置文件是 Prometheus 中比较复杂但十分重要的部分，这里进行过于详细的介绍，读者可以在它的官方文档中找到关于这个配置文件内容的十分详细的说明^①。

另一个比较重要的配置是性能指标的仪表盘，也就是 Grafana 中的 Dashboard。值得庆幸的是，这部分的操作主要通过在界面单击、拖曳就可以完成。其中比较值得一说的是指标查询的部分，例如在 Dashboard 添加一个图表区域后，在图表的编辑界面有一块需要用

① <https://prometheus.io/docs/operating/configuration/>

户配置的指标查询表达式，如图 7-10 所示。

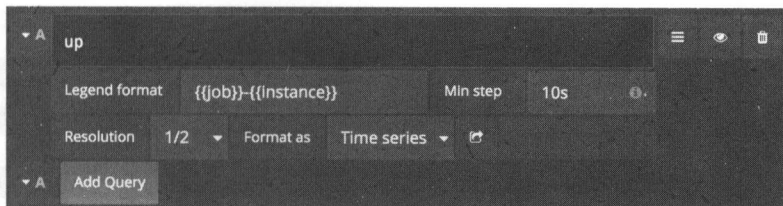


图 7-10 Grafana 的数据查询配置

Prometheus 定义了专用的查询 DSL（称为 PromQL）用于从它的数据库中选择所需的数据^①。以下举一些比较实用的节点和容器监控的指标例子。

1. 采集器本身的指标数据

采集器的运行状态（Bool）：

```
up{job="采集器类型"}
```

2. 节点（NodeExporter）指标数据查询

节点运行时长（s）：

```
node_time - node_boot_time
```

节点过去 1 分钟 / 5 分钟 / 15 分钟的平均负载：

```
node_load1 或 node_load5 或 node_load15
```

指定磁盘的剩余空间（Byte）：

```
node_filesystem_avail{device="/dev/xvda1",mountpoint="/rootfs"}
```

指定磁盘的已用空间（Byte）：

```
node_filesystem_size{device="/dev/xvda1",mountpoint="/rootfs"} -  
node_filesystem_avail{device="/dev/xvda1",mountpoint="/rootfs"}
```

指定磁盘的剩余空间百分比（%）：

```
node_filesystem_avail{device="/dev/xvda1",mountpoint="/rootfs"} /  
node_filesystem_size{device="/dev/xvda1",mountpoint="/rootfs"}
```

节点空闲内存（Byte）：

^① <https://prometheus.io/docs/prometheus/1.8/querying/basics/>

`node_memory_MemFree`

节点可用内存（空闲内存+Buffer+Cache）(Byte)：

`node_memory_MemAvailable`

节点的 CPU 使用百分比 (%)：

`(1 - avg(irate(node_cpu{mode="idle"}[30s])) by (instance)) * 100`

节点已建立的 TCP 连接数：

`node_netstat_Tcp_CurrEstab`

节点网络 I/O 流入 / 流出量 (Byte)：

`node_netstat_Tcp_InSegs` 或 `node_netstat_Tcp_OutSegs`

磁盘实时 I/O 读 / 写量 (页，即 512Byte)：

`node_vmstat_gpgpin` 或 `node_vmstat_gpgpout`

3. 容器 (cAdvisor) 指标数据查询

容器的磁盘占用 (Byte)：

`avg(container_fs_usage_bytes{id=~"/docker/.*"}) by (instance,name)`

容器的可用磁盘上限 (Byte)：

`avg(container_fs_limit_bytes{id=~"/docker/.*"}) by (instance,name) -
avg(container_fs_usage_bytes{id=~"/docker/.*"}) by (instance,name)`

容器的内存占用 (Byte)：

`avg(container_memory_usage_bytes{id=~"/docker/.*"}) by (instance,name)`

容器的 CPU 使用百分比 (%)：

`avg(irate(container_cpu_user_seconds_total{id=~"/docker/.*"}[30s]) * 100) by (instance,name)`

在实际情况下，集群中的容器数量和种类可能非常多（许多是临时的任务容器、基础设施的容器等），而实际运营所关心的容器只是其中的一部分。因此通常还需要加上 `name` 的正则表达式或其他标签，对图表中展示的内容进行进一步过滤，防止因图表信息过载而淹没真正有用的信息。

除了指标本身的选择，默认情况下绘制图表的图例（即解释每条曲线意思的文字）使用的是所选指标的完整名称，这个名称通常很长且难以阅读。可以在配置界面的 **Legend**

`format` 属性里修改显示的图例内容。

节点通常用实例的采集来源（IP 地址或域名）作为图例，在图例格式属性中指定，如下所示。

```
{{instance}}
```

对容器的定位比较有用的信息包括容器的名称、镜像以及所在的节点，常用的图例格式如下所示。

- `{{name}}`
- `{{instance}} - {{name}}`
- `{{instance}} - {{image}} - {{name}}`

图 7-11 展示了一个配置完成的节点和容器性能数据面板。

PromQL 中内置了很多的运算和操作函数。这些函数的命令大多比较符合直觉，在文档里的介绍也比较清晰。唯独其中的 `rate` 和 `irate` 两个函数比较容易弄混，值得单独说明一下。这两个函数在大多数情况下的效果差不多，可以相互替换。它们的细微差异在于，`rate` 函数获得的是指定范围内的记录平均每秒的变化值，通常用于告警和展示变化比较缓慢的数据，可以过滤偶尔的峰值噪声。而 `irate` 函数总是用相邻的两个记录计算每秒的值，通常用于展示变化十分迅速且不希望丢失其中任何一次峰值的情况指标。

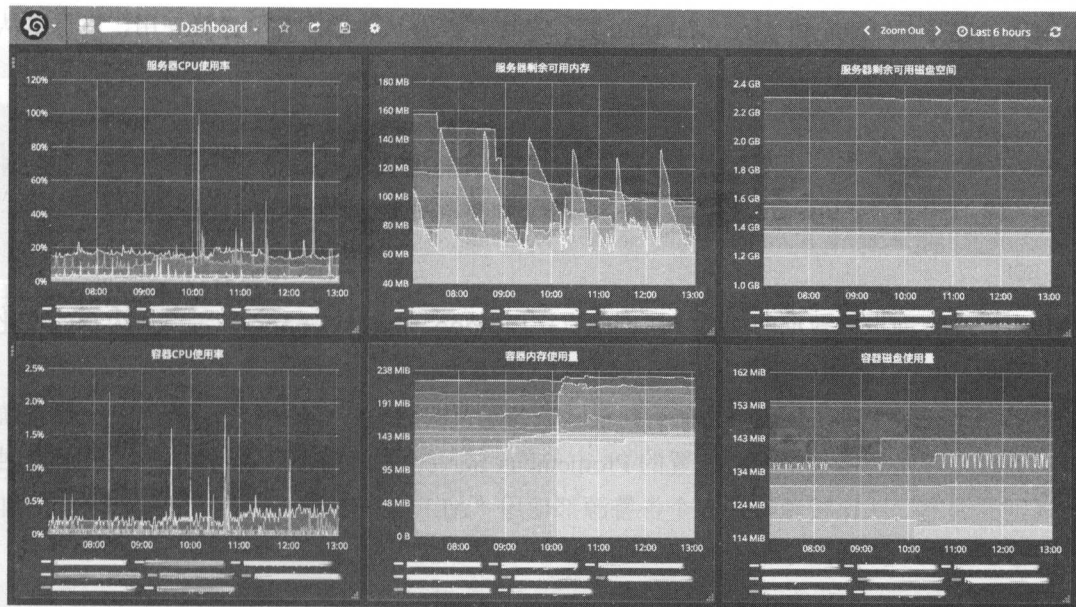


图 7-11 监控节点和容器的资源消耗情况

上个小节还部署了 PushGateway 组件，并介绍了它可以将推送型的数据源变成 Prometheus 需要的拉取型数据源。和 Prometheus 配置文件中对采集过程的管理方法一样，每个指标都应该属于某个采集任务 (job)、每个任务可以有一个或多个来源实例 (instance)。为了使 PushGateway 上采集数据，同样需要 PushGateway 服务的地址添加到 “prometheus.yml” 配置文件中，可以使用静态或动态服务发现的目标指定方法。

相比 Prometheus 的部署，它的配置方面略显复杂，但由于其庞大的社区资源，Prometheus 的监控范围已远不局限于基础设施方面，对于常用的服务组件（数据库、缓存、队列等）都有指标丰富的采集工具。因此一旦熟悉上手便可使用几乎相同的方法快速将整个集群中的各种目标资源监控统一地监控起来。

最后来说说规则文件使用。在介绍 “prometheus.yml” 主配置文件时提到了 `rule_files` 这个属性，它的值是一个列表，其中的每一项都是一个规则文件。目前 Prometheus 只支持两种类型的规则文件：“告警触发规则” 和 “数据记录规则”。

`rule_files` 属性的值可以有任意多条，用户并不需要显式地区分不同的规则文件类型，因为它们都会在每次数据采集完成后被执行，且各自具有不同的内容格式。在实例的 “prometheus.yml” 里指定了两个规则文件，现在就来创建这些规则。

首先是数据记录规则 `record.rules`，在这个文件中可以额外定义一些组合指标，这些指标会自动在每次采集数据后，在 Prometheus Server 中自动计算出来。从而在系统的各个组件（包括 Grafana 和 Alert Manager 里）中将它作为一种指标类型直接使用，就像是从原始目标采集来的一样，如下所示。

```
$ cat <<EOF >/etc/prometheus/rules/record.rules
node_memory_releasable = node_memory_Buffers + node_memory_Cached
node_disk_mb_read = node_disk_bytes_read / 1048576
node_memory_rate = sum(rate(node_memory_MemFree[5m])) by (job)
node_leak_memory = count(node_memory_MemFree < 1000000)
EOF
```

在任何查询指标数据的地方，都可以直接用 `node_memory_rate` 表示节点内存的变化率、用 `node_disk_mb_read` 表示以 mb/s 计数的磁盘每写入的速度等。

另一个记录文件 `alert.rules` 则表示告警触发的规则。已经提到过 Prometheus 系统的告警触发规则是由处于核心位置的 Prometheus Server 生成的。Prometheus Server 告警触发规则的编写语法比较独特，每个告警源都使用 “ALERT” 加上一个告警的名称开始，如下所示。

```
$ cat <<EOF >/etc/prometheus/rules/alert.rules
ALERT InstanceDown
  IF up == 0
```

```

FOR 5m
LABELS { severity = "page" }
ANNOTATIONS {
    summary = "Instance {{ $labels.instance }} down",
    description = "{{ $labels.instance }} of job {{ $labels.job }} has been down
for more than 5 minutes.",
}

ALERT MemoryHigh
IF prometheus_local_storage_memory_series > 200
FOR 1s
ANNOTATIONS {
    summary = "Prometheus using more memory than it should
{{ $labels.instance }}",
    description = "{{ $labels.instance }} has lots of memory man (current
value: {{ $value }}s)",
}
EOF

```

这里定义的两条告警规则分别表示“有节点出现故障崩溃”和“节点的剩余内存不足”两种事件。关于这几个文件的配置细节，同样可以在 Prometheus 文档的相应章节找到。

修改完 Prometheus Server 的配置文件，不用重启整个 Prometheus Server 服务，只需找出它进程的 PID，然后给这个进程发一个 SIGHUP 信号，就可以使其重新加载配置文件了，如下所示。

```
$ kill -HUP <PID>
```

有了告警规则，Prometheus 就会在匹配到告警事件时给 Alert Manager 发出信息了（实际上是调用 Alert Manager 服务的“/api/v1/alerts”接口）。上个小节部署了 Alert Manager，在它的配置里添加了一条处理规则，即把所有告警转发到所在节点的 5001 端口上的另一个服务。现在来实现这个监听 5001 端口的 Python 脚本，它会将所有告警传递的数据打印到控制台，如下所示。

```

$ pip install Flask
$ cat <<EOF | sudo tee /usr/lib/handler.py
import sys
import json
from flask import Flask, request, jsonify
app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def echo():
    try:

```

```
    json_obj = request.get_json()
    sys.stderr.write(json.dumps(json_obj, indent=2, sort_keys=True))
except Exception as e:
    sys.stderr.write(jsonify(message=e))
return ''

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5001)
EOF
```

在后台启动这个处理脚本待用，如下所示。

```
$ (python /usr/lib/handler.py >/tmp/webhook.log 2>&1 &)
```

然后随意停掉集群中被监控的一个节点上的 Node Exporter，稍等几秒钟，就会在“/tmp/webhook.log”这个日志文件里看到相应的告警信息。

Prometheus 的设计基于大量实际用户的场景和反馈，已经算是比较完善了。但需要指出的是 Prometheus 并非毫无缺陷，在当前的 Prometheus 1.x 版本中有两个需要特别注意的问题。

首先是高可用的问题，目前 Prometheus 的核心组件 Prometheus Service 还不能支持高可用的部署方式，因此存在潜在的单点故障。这是由于 Prometheus 在初期设计时将采集数据的处理和存储都放在了 Prometheus Service 组件里，使得这个组件无法直接进行水平扩容。官方正在着手解决此问题，将在 Prometheus 2.0 版本中提供单独的时间序列数据库组件，从而实现数据在线备份和服务高可用等特性。

其次是集群规模的问题，Prometheus 的数据拉取模式（即所有数据的采集由中心节点发起，而不是各个采集器主动推送）带来了很方便的集中式配置体验，但同时使得作为所有采集请求发起方的 Prometheus Server 节点工作负载较高，这一点在集群规模较大（大于 1000 节点，若在每个节点采集的数据量较多，甚至只能采集更少的节点）时体现得尤为明显。典型的特征是中心节点占用大量 CPU 和内存资源，进行数据查询时的速度也非常缓慢，甚至超时。在 Prometheus 1.6 版本过后，这个现象已经有了比较大的改善，不过早期 Prometheus 为了解决这个问题，在架构中支持 Server 节点的级联，如图 7-12 所示。这个结构在大规模集群、无法提供高配置主机以及希望对不同基础设施类型进行分别监控然后汇总的场景中依然十分具有参考意义。

位于上层的 Federation Prometheus 与普通的 Prometheus Server 并没有本质区别，只不过它所采集的目标不是单个 Exporter，而是其他的 Prometheus Server 通过级联 API（HTTP 路径为“/federate”）汇聚后的批量 Exporter 数据。通过这样的结构，只需两级 Prometheus

便可满足十万级以上节点规模数据中心的性能数据汇聚。

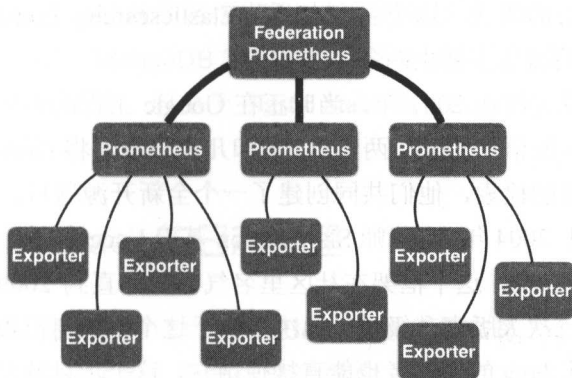


图 7-12 Prometheus 的级联结构

7.2 集群日志管理

7.2.1 常见的开源日志管理方案

集中管理日志对于比较大型的服务集群而言绝对是防患于未然的必要措施，即使对于规模不太大的服务集群，它也是定位系统故障以及分析服务运行数据的好帮手。

诞生于 2003 年的 Splunk 曾经一度是企业级日志收集管理领域中最成功的一款端到端解决方案之一，它将分散在各个地方的各式各样的日志统一采集上来，将这些日志内容进行分类、索引并加上与内容相关的标签，使得用户可以像搜索引擎那样在海量的日子中快速地检索到寻找的内容。从整体流程上看，Splunk 对日志的处理可以分成原始日志采集、日志内容归档、信息提取以及规则化查询和展示这几个环节。这个过程似乎并不复杂，但这种对实时性要求极高的业务，只要数据规模提上来，任何简单的事情就将不再简单。

虽然 Splunk 功能强大，也并非所有的企业都愿意负担它的高昂的价格（有免费版本，但对功能和数据量有限制，不适合正规企业使用）。在开源世界里，即使早在 2001 年的时候就有了 Lucene^①这样的全文检索引擎项目，但第一个真正能挑战 Splunk 地位的开源日志解决方案 ELK 一直到 2013 年以后才算基本成型，这个方案也经历了很长的一段曲折故事。

^① <https://github.com/apache/lucene-solr>

ELK 是 Elastic 公司推出的开源日志收集产品，现在已经改名为 Elastic Stack 技术栈，ELK 是其 5.0 版本之前的叫法，这个名字来源于 Elasticsearch、Logstash 和 Kibana 三个项目组合的缩写。ELK 的诞生主要由两条主线构成。

第一条主线的开端大约在 2007 年，当时正在 Google 工作的工程师 Jordan Sissel 创造了一种新的正则模式匹配语法 Grok。两年后，他和几位开发者将 Grok 与 Lucene 结合起来，开始尝试将它用于日志的检索，他们共同创建了一个全新开源项目：Logstash。

第二条主线开始于 2004 年，工程师 Shay Banon 基于 Lucene 项目创造了一款 Java 开发搜索引擎的框架：Compass，这个框架在社区里名气甚好。直到 2009 年，当 Shay 准备对 Compass 框架进行第三次大版本升级时，他决定基于这个框架自己设计一款更加完整的引擎的解决方案，使得非 Java 的开发者也能直接使用它，这个项目就是后来的 Elasticsearch。新的项目为 Shay 带来了前所未有的商机，Shay 在 2012 年创建了自己的公司 Elasticsearch BV，并将 Logstash 项目购入旗下，开始正儿八经地经营以日志管理和分析为核心的生意。2013 年，Kinana 项目诞生，作为日志分析和搜索结果展示的 Web 前端，至此 ELK 这套组合拳才算是凑齐了。

ELK 推出后发展迅猛，在 2014 年的 Google 搜索趋势指数上，ELK 就开始赶超 Splunk^①，并逐渐成为许多企业日志处理的首选方案。不过，ELK 作为日志采集端的 Logstash 是使用 JRuby 编写的，运行时对资源的消耗比较高，而采集端是要在集群的每个节点上全面部署的，这就会对集群的性能产生一定影响。为了规避这个问题，一个新的基于 Golang 的数据采集器被创造了出来，称为 Beats。它其实是几种类型的数据采集器的总称，采集的范围也已经不限于日志，具体有细分为收集文件日志的“FileBeat”、收集 Windows 系统日志的“Winlogbeat”、收集性能数据的“Metricbeat”、收集网络数据的“Packetbeat”和检查服务运行状态的“Heartbeat”。这套新的采集工具在 2015 年底正式发布了 ELK 1.0 版本。由于这个新成员的加入，官方也将原本的“ELK Stack”解决方案更名为“Elastic Stack”。同时 Elastic 公司将过去的多款涉及数据安全、数据分析、数据告警等企业级付费插件整合成了一个独立组件“X-Pack”，它并不是开源的，需要成为 Elastic 企业级套餐的订阅用户才能长期使用，普通用户可以安装 30 天免费试用版本。

那么 Elastic Stack 是不是当前开源界日志处理的唯一方案呢？显然不是，曾经在 Linux 运维界流行过的 RSyslog、Facebook 开源的 Scribe、Apache 基金会旗下的 Flume 以及 Fluentdata 公司开源的 Fluentd 等都在尝试做类似的事情。本书选择其中的 Fluentd 作为另一个重点介绍的方案，原因很简单，Fluentd 是 CNCF 基金会最早的项目之一，与 Kubernetes

① <http://www.infoworld.com/article/2998136/open-source-tools/is-open-source-overtaking-splunk.html>

和 Prometheus 可以说是师出同门，自带 Cloud Native 光环，与容器技术的集成也比较好。Fluentd 可以作为 Beats 和 Logstash 的替代组件，在存储和展示方面，Fluentd 通过插件机制支持接入 Solr、Elasticsearch、MongoDB 等多种组件以及相应的展示平台，也可接入 Kafka 直接进入数据分析平台进行处理。但由于 Elastic Stack 组件的流行，目前 Fluentd 最常见的组合方案还是 Elasticsearch 和 Kibana。

7.2.2 基于 Elastic Stack 的日志管理

Elastic Stack 的服务结构如图 7-13 所示。Beats 会被部署在每一个节点上，它是一个十分轻量的工具，不仅占用的 CPU 和内存资源很少（例如空载时 Filebeat 内存消耗约 10~15MB，CPU 消耗可忽略不计），部署起来也很容易。Beats 可以将数据直接发送到 Elasticsearch 服务中存储，也可以先发送给 Logstash，然后在 Logstash 上进行初步的分析，以及内容的转换和标注操作。Logstash 不具有数据持久化的功能，最终还是要将处理过后的内容发送到外部存储引擎（通常是 Elasticsearch）。Elasticsearch 是数据存储以及进行检索、分析的主要地方，本质上是一个全文搜索引擎。Kibana 是一个为终端用户进行日志内容检索的 Web 界面服务，可以根据 Logstash 的标注以及日志文本的内容进行过滤和查找，并通过列表或图形的方式将结果展示出来。

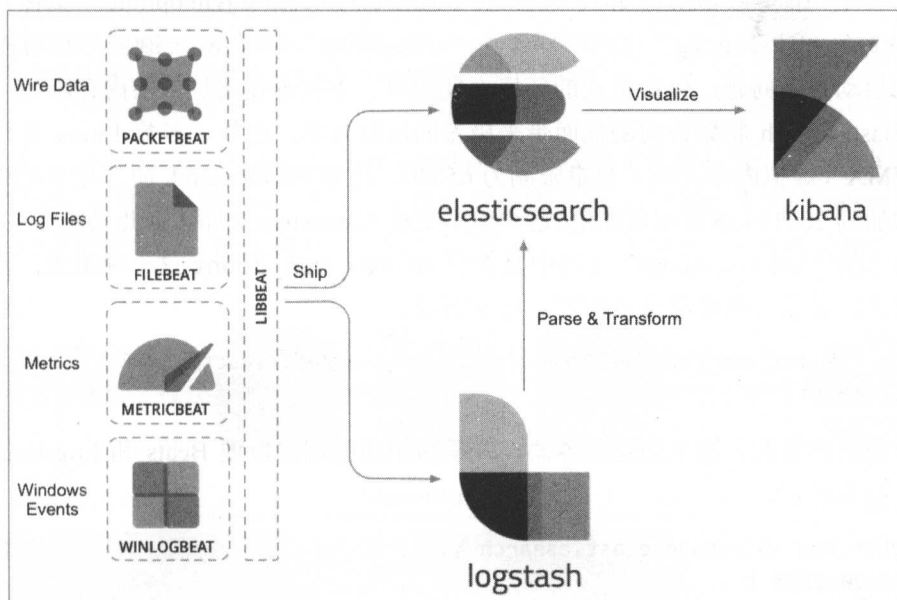


图 7-13 Elastic Stack 的服务整体结构

其中对于日志的采集，通常只需要使用 Filebeat 服务即可（Windows 服务器上还会用到 Winlogbeat）。使用 Golang 编写的 Filebeat 只有一个二进制文件，放到服务器上，然后指定正确的参数和配置文件运行即可。不过为了便于读者移植到集群中进行容器化的自动部署，这里统一介绍采用容器方式进行部署的方法。

Elastic Stack 服务的官方容器镜像都不在 Docker Hub 中，而是存放在 Elasticsearch BV 公司独立的镜像仓库。其镜像的源代码都在 GitHub，因此不论是想对镜像内容一探究竟还是想基于官方镜像进行二次开发都不困难。相应的源码仓库地址如下所示。

- <https://github.com/elastic/elasticsearch-docker>。
- <https://github.com/elastic/logstash-docker>。
- <https://github.com/elastic/beats-docker>。
- <https://github.com/elastic/kibana-docker>。

最先需要启动的是作为存储中心的 Elasticsearch 服务。除了文本数据的存储功能，Elasticsearch 还是一个高速文本搜索引擎，因而需要使用大量内存缓存数据的全文索引，官方推荐预留至少 2GB 内存。实际上，在默认情况下，Elasticsearch 启动时就要创建一块 2GB 大小的独占内存池，如果节点当前剩余的内存不足 2GB，就会出现“failed to map reserved memory”的错误提示。

Elasticsearch 的 JVM 配置文件（通过 deb 或 rpm 包安装的是“/etc/elasticsearch/jvm.options”文件，官方容器里则是“/usr/share/elasticsearch/config/jvm.options”文件）将默认的“-Xms2g”和“-Xmx2g”这两行修改为“-Xms500m”和“-Xmx500m”后可以解决内存不足的启动报错问题，但在正式的环境中使用，不推荐使用过低的内存限制，这可能导致因 Elasticsearch 频繁访问磁盘而带来极差的运行效率。此外，多数 Linux 发行版的默认进程 VMA（虚拟内存区域）数量限制为 65530，这对 Elasticsearch 而言是不足够的，需要将其增加到 262144 或者更高的值。这个值不支持 Namespace 隔离，因此只能在主机修改。检查节点上的“/etc/sysctl.conf”文件中是否有 vm.max_map_count 这一项配置，若有可将其数值增大，若没有则增加这个配置，如下所示。

```
$ echo 'vm.max_map_count=524288' | sudo tee -a /etc/sysctl.conf
$ sudo sysctl -p
```

使用完整的镜像名称来创建此容器，并暴露相应的端口以便 Beats 和 Logstash 服务能连接它，如下所示。

```
$ docker run -d --name elasticsearch \
  -p 9200:9200 \
  -p 9300:9300 \
  docker.elastic.co/elasticsearch/elasticsearch:5.4.0
```

然后启动 Logstash 服务。Logstash 是日志进行集中处理和转发的地方，在这里可以通过各种插件扩展日志的来源、过滤转换方式以及输出位置。实际上 Beats 只是 Logstash 支持的几十种数据来源的其中一种，它还可以收集来自消息队列、本地文件，甚至 GitHub 提交记录和 Twitter 消息记录等第三方平台的输入。日志的过滤和转换插件也有几十种，其中“grok”“drop”和“geoip”等经典插件几乎从最早版本的 Logstash 就已经存在。Grok 是一种将非结构化数据进行模式匹配，从而提取关键内容的语法，关于 Grok 语法的细节在许多专门介绍 Logstash 的书籍和文章都有介绍，这里不再展开说明，官方提供了一个在线的语法调试工具^①。

从 5.0 版本开始，Logstash 的主要配置文件是“logstash.yml”。官方镜像中默认安装了 30 天试用版的 X-Pack 服务，X-Pack 提供基于日志内容进行监控和告警功能，这会导致需要一些额外的配置。镜像中内置的“logstash.yml”中关于 X-Pack 的配置文件无法直接使用，需要基于其镜像代码仓库的源码进行修改，或在启动时挂载一个文件将其覆盖。考虑到它只有 30 天试用期，建议是直接将它从镜像中去除。这里为了尽可能地采用官方镜像进行介绍，采用启动时覆盖配置的方法，先创建一个名为“logstash.yml”的文件，填上正确的配置内容（假设 Elasticsearch 的节点 IP 地址为 172.31.19.147），如下所示。

```
$ cat <<EOF | sudo tee >/dev/null /etc/logstash/logstash.yml
http.host: "0.0.0.0"
path.config: /usr/share/logstash/pipeline
xpack.monitoring.elasticsearch.url: http://172.31.19.147:9200
xpack.monitoring.elasticsearch.username: elastic
xpack.monitoring.elasticsearch.password: changeme
EOF
```

其中的“path.config”配置的是存放 Logstash 处理规则文件的目录。Logstash 处理规则使用一种与 Json 略相似的 DSL 语法，主体内容分成“input”“filter”和“output”三个部分，分别对应数据来源、过滤和输出。图 7-14 展示了一个典型的日志数据处理过程。

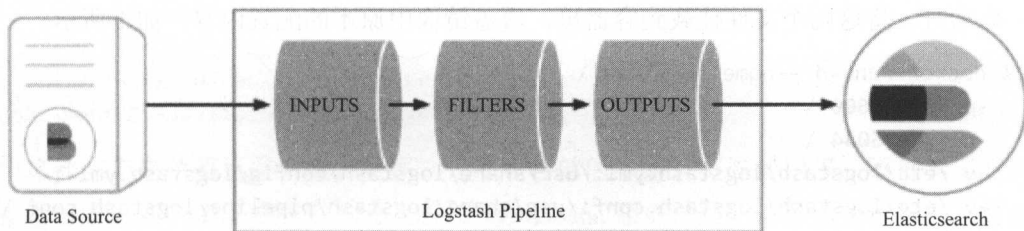


图 7-14 Logstash 日志数据处理过程

^① <http://grokdebug.herokuapp.com/>

以下配置指定了一个 Beats 数据源，并将来源目录是“/log/demo”的日志内容进行字段分解，最后输出处理后的内容到 Elasticsearch 服务。其中对日志内容的分解采用了一段 Grok 表达式，它的前提是该目录下的日志文件都符合特定的输出规则，这个例子假设每行日志内容均以“IP 地址 HTTP 方法 HTTP 路径 [日志级别] 日志内容”这种输出格式。例如“10.12.45.234 GET /check [ERROR] database not ready”，它将依据预设的各部分含义被分解成多个字段分别存储，以便将来进行检索。

```
$ cat <<EOF | sudo tee >/dev/null /etc/logstash/logstash.conf
input {
  beats {
    port => "5044"
  }
}
filter {
  grok {
    patterns_dir => [ "/log/demo/" ]
    match => { "message" => "%{IP:client} %{WORD:method} %{URIPATHPARAM:request}
\[%{WORD:level}\] %{GREEDYDATA:message}" }
    add_field => [ "received_from", "%{host}" ]
    overwrite => [ "message" ]
  }
}
output {
  elasticsearch {
    hosts => [ "172.31.19.147:9200" ]
    user => "elastic"
    password => "changeme"
  }
}
EOF
```

启动时，将这两个文件挂载到容器里，覆盖镜像中原本的配置内容，如下所示。

```
$ docker run -d --name logstash \
  -p 9600:9600 \
  -p 5044:5044 \
  -v /etc/logstash/logstash.yml:/usr/share/logstash/config/logstash.yml \
  -v /etc/logstash/logstash.conf:/usr/share/logstash/pipeline/logstash.conf \
  docker.elastic.co/logstash/logstash:5.4.0
```

接下来部署 Filebeat。作为日志的采集器，Filebeat 被设计得十分轻量，同时它需要被部署到每一个有日志产生的节点上。在每个节点上的 Filebeat 配置可以不一样，但为了方

便管理，通常会按照节点的用途使用脚本和模板生成这些配置。配置内容采用 YAML 格式，主要配置项与 Logstash 相似但更加简单，包括采集日志的目录列表、可选的内容过滤规则和输出的目标^①。下面的这个例子将采集当前主机上的所有容器产生的日志以及一个指定目录下的日志内容，然后仅保留其中包含 “[ERROR]” 或 “[WARNING]” 的行，并转发到运行在 172.31.20.128 节点上的 Logstash 服务。

```
$ cat <<EOF | sudo tee >/dev/null /etc/filebeat/filebeat.yml
filebeat.prospectors:
- input_type: log
  paths:
    - /log/demo/*.log
    include_lines: ['\[ERROR\]', '\[WARNING\]']
- input_type: log
  paths:
    - /containers/*/*-json.log
    include_lines: ['\[ERROR\]', '\[WARNING\]']
output.logstash:
  hosts: ["172.31.20.128:5044"]
EOF
```

运行 Filebeat 容器，将节点的 “/var/lib/docker/containers” 目录挂载到容器里，如下所示。默认情况下，当前节点的所有 Docker 容器日志都会以 Json 格式存放在这个目录下以容器 ID 命名的子目录里，因此从这里就可以采集到所有容器输出的内容。值得指出的是，Docker 存放日志的目录只有 root 用户有权限访问，因此要采集容器的日志，要么将这个目录的访问权限放宽，要么使用 root 用户启动容器。另外还需要挂载其他存放日志的目录到容器中，比如 “/var/log” 这个目录经常被作为系统服务的默认日志存储位置。

```
$ docker run -d --name filebeat \
-v /etc/filebeat/filebeat.yml:/usr/share/filebeat/filebeat.yml \
--user root \
-v /var/log:/log:ro \
-v /var/lib/docker/containers:/containers:ro \
docker.elastic.co/beats/filebeat:5.4.0
```

这样一个简单的日志采集、汇聚、处理和存储的系统就可以使用了。不过为了更直观地检索日志内容，还需要一个 Web UI。启动 Kibana 服务，将它的数据来源指向之前配置好的 Elasticsearch 服务，并暴露 5601 端口到节点上，如下所示。

① <https://www.elastic.co/guide/en/beats/libbeat/5.x/config-file-format.html>

```
$ docker run -d --name kibana \  
-p 5601:5601 \  
-e ELASTICSEARCH_URL=http://172.31.19.147:9200 \  
-e ELASTICSEARCH_USERNAME=elastic \  
-e ELASTICSEARCH_PASSWORD=changeme \  
docker.elastic.co/kibana/kibana:5.4.0
```

例子中的“ELASTICSEARCH_URL=http://172.31.19.147”参数表示 ElasticSearch 服务运行的地址，应该根据实际情况提供。

打开浏览器，访问 Kibana 服务所在节点的 5601 端口，就会打开 Kibana 的登录页面。输入 Elasticsearch 服务的用户和密码，默认是 elastic/changeme。

第一次进入 Kibana 时，由于 Elasticsearch 中还没有任何来自 Logstash 的数据，界面上创建索引配置的按钮是灰色的，如图 7-15 所示。现在来制造一些日志数据，例如在任意部署了 Filebeat 的节点上执行，如下所示。

```
$ echo '127.0.0.1 POST /failed [WARNING] This is a message' \  
>> /var/log/demo/hello.log
```

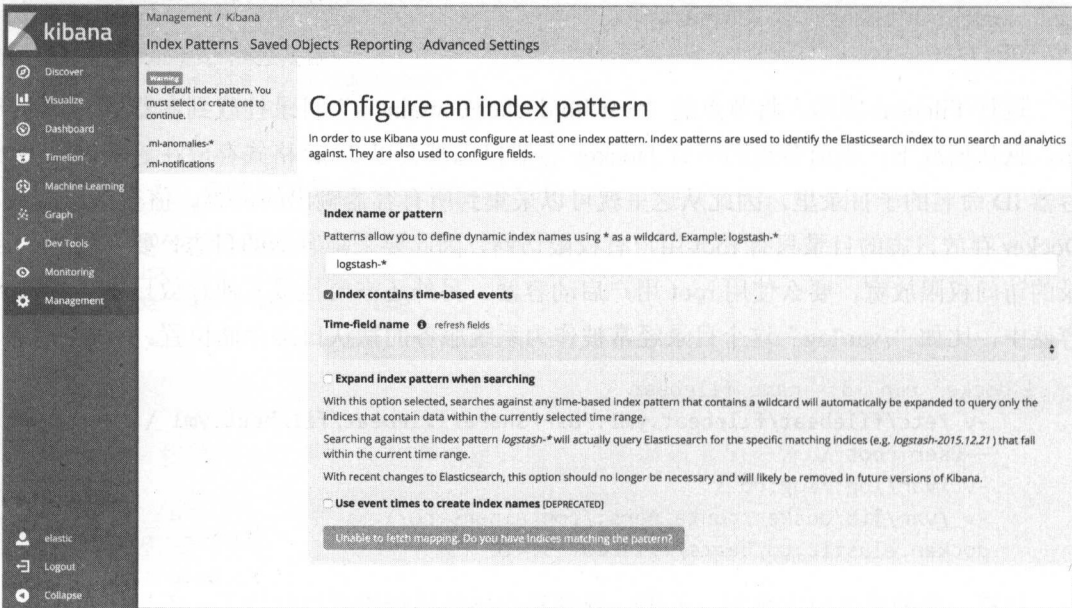


图 7-15 Kibana 的创建数据索引页面

这个命令会在指定的“/var/log/demo/”目录下创建一个新的文件，并写入一些符合模板规则的内容。或是创建一些会在控制台打印内容的容器，例如以下命令使用 flin/go-echo 镜像创建一个用于输出日志的容器，它会将收到的所有 POST 消息内容回显到控制台，这

些写到控制台的日志会以 Json 格式保存到 Docker 的安装目录下, 然后被 Filebeat 所采集到。

```
$ docker run -d --name echoes -p 8000:8000 flin/go-echo
```

访问映射到本地的 8000 端口, curl 会展示请求返回的内容, 这些内容也会出现在容器的日志里, 如下所示。

```
$ curl -XPOST -d '[ERROR] Error from container' localhost:8000
```

在 Elasticsearch 里汇聚到一些来自 Logstash 的日志内容后, 刷新 Kibana 界面, 按钮上的文字变成 “Create”, 单击按钮即可完成索引的配置。然后单击左侧菜单栏的 “Discover” 进入日志搜索页面。

若在创建日志后, Kibana 的界面上依然显示找不到可用数据。可以沿着数据来源的路线寻找一下线索。首先检查日志文件所在节点上的 Filebeat 服务, Filebeat 没有提供 API, 但可以从它自身的日志看到一些信息, 如下所示。

```
$ docker logs -f --tail=20 filebeat
```

默认配置下 Filebeats 每隔 30s 会向日志里写入一条采集到的日志数量等信息, 如果看到 “INFO Non-zero metrics in the last 30s: ...” 这样的内容, 表示数据已经被 Filebeat 采集到, 可以比较一下写入日志的行数和统计的数目是否一致。

若 Filebeats 正常, 接下来检查 Logstash。假设 Logstash 服务运行在地址为 172.31.20.128 的节点上, 它提供了一组状态查询的 API, 例如查看汇聚的数据统计信息, 如下所示。

```
$ curl -s 'http://172.31.20.128:9600/_node/stats/pipeline?pretty=true'
```

输出的结果包含经过每一个处理步骤后通过的日志条目数, 在这里可以检测是否因为配置的不正确而使日志被意外过滤掉。若最终输出的条目数大于 0, 则接下来检查 Elasticsearch 里是否写入了相应的记录。

Elasticsearch 有一套十分强大的内容搜索 API, 不仅支持复杂的聚合和逻辑嵌套, 还具备全文模糊查询的功能, 能处理日志内容中的拼写错误、语序颠倒等情况。本书不会详细介绍这些 API, 读者可以在 Elasticsearch 的文档中找到更多信息^①。Elasticsearch 存储的数据是按照索引 (Index) 存储的, Logstash 汇聚数据时, 默认使用 logstash-加上日期作为日志的索引。下面这个查询可以找到所有 Logstash 存储到 Elasticsearch 的数据。

```
$ curl -s 'http://elastic:changeme@172.31.19.147:9200/  
logstash-*/_search?pretty=true&q=*:*
```

^① <https://www.elastic.co/guide/en/elasticsearch/reference/current/search.html>

刚刚在 Kibana 添加的检索配置也使用了 `logstash-*` 这个索引匹配关键字，这样确保之后的搜索都是在 Logstash 采集到的数据中进行的。

每行存储在 Elasticsearch 中的日志文本都被加上了许多标签，这些标签被称为“字段 (Field)”，例如来源地址、采集时间、采集器版本等，在 Logstash 的日志处理配置里也对原始的日志内容进行了进一步拆解，使之成为有独立业务含义的多个字段。以下的内容就是从“hello.log”文件中采集到的一行日志存储到 Elasticsearch 后的样子。

```
{
  "_index" : "logstash-2017.08.18",
  "_type" : "log",
  "_id" : "AVy7807GVuQj7XKuPx3D",
  "_score" : 1.0,
  "_source" : {
    "request" : "/failed",
    "offset" : 153,
    "method" : "POST",
    "level" : "WARNING",
    "input_type" : "log",
    "source" : "/log/demo/hello.log",
    "message" : "This is a message",
    "type" : "log",
    "tags" : [
      "beats_input_codec_plain_applied"
    ],
    "received_from" : "04146e036a45",
    "@timestamp" : "2017-06-18T16:04:57.938Z",
    "@version" : "1",
    "beat" : {
      "hostname" : "04146e036a45",
      "name" : "04146e036a45",
      "version" : "5.4.0"
    },
    "host" : "04146e036a45",
    "client" : "127.0.0.1"
  }
}
```

Kibana 仅支持 Elasticsearch 庞大查询语法中的一部分子集，称为“字符串查询语法 (Query String Syntax)”^①。下面介绍这种语法的常用搜索规则。

① <http://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html#query-string-syntax>

只给定一个搜索词，在所有记录的“默认字段”（对于日志而言是“message”字段）中搜索匹配的内容，例如下面这个搜索会找到日志内容中包含“This”的结果。

```
This
```

默认使用空格分隔搜索词，若搜索词本身包含空格，可以用英文引号（“”）将搜索内容包裹起来，例如下面这个搜索会找到日志内容中包含“This is”这个完整词组的结果。

```
"This is"
```

而下面这个搜索则会找到日志中包含“This”或包含“is”的记录。

```
This is
```

在搜索词前面加上加号（+）或减号（-）表示必须存在或必须没有的内容，例如下面这个搜索会找到同时包含“This”和“is”单词，但是不包含“are”单词的结果。

```
+This +is -are
```

没有前缀符号的搜索词实际上表示：如果存在则提高匹配度，但不一定必须有（这也解释了将多个无前缀的关键词并列在一起时，任意一个词存在都可以匹配上的原因）。例如下面这个搜索会找到内容一定含有“This is”关键字，最好有“a”或者“an”，但是不能有“are”的记录。

```
+"This is" -are a an
```

使用“字段：搜索词”的格式在所有记录的指定字段中进行搜索。例如下面这个搜索会匹配所有请求 URI 是“/failed”的记录。

```
request: "/failed"
```

之前例子中的搜索相当于在“message”字段中进行搜索，如下所示。

```
message:(+"This is" -are a an)
```

支持 AND、OR、NOT 逻辑组合。例如下面的规则会找到所有请求 URI 是“/failed”且内容包含“This”的记录。

```
This AND request: "/failed"
```

这种逻辑运算遵循通常的“非>与>或”的运算优先级关系，可以用括号来改变运算的顺序。

使用问号（?）或星号（*）作为单个字母或任意一个字母的通配符。例如搜索“Th*”或“Th??”都会匹配到单词“This”。

使用特殊的 “_exists_” 关键字标示特定字段存在，例如下面的规则会匹配所有包含 “method” 字段的记录。

```
_exists_:method
```

使用中括号和关键字 “TO” 表示数值或时间范围，例如下面这个规则匹配采集时间是 2017 年 1 月至 12 月之间的日志。

```
@timestamp:[2017-01-01T00:00:00 TO 2017-12-31T23:59:59]
```

使用波浪号 (~) 表示模糊匹配，例如使用 “msesaeg~” 或是 “mesage~” 这样的错误拼写都能匹配到 “message” 这个词。可以在波浪号后面加上一个数字，表示能够接受的字母修改个数，如 “msesaeg~1” 就不能匹配 “message”，因为有两个单词错序。波浪号更常用的场景是在句子中，表示在指定句式的单词之间可以相距一定的距离，例如 “This message~2” 会匹配到 “This is a message” 这样的日志内容。

使用上尖括号 (^) 表示匹配权重，例如 “This^2 is^1” 表示搜索包含 “This” 或 “is” 的行，同时优先查找有 “This” 的结果，即只包含 “This” 的结果会在只包含 “is” 的结果行的前面。

通过 Kibana 的界面还能对日志显示的标签进行精简、修改搜索时间范围以及通过日志搜索结果建立监控看板等许多功能。这里不进一步展开，读者可自行探索。

7.2.3 基于 Fluentd 的日志管理

Elastic Stack 功能的强大是毋庸置疑的，不过随着 X-Pack 等服务的推出，Elasticsearch BV 公司的商业化倾向性日益明显，从官方容器镜像迁移至私有仓库，以及在社区镜像中集成试用版的 X-Pack 服务的运作手段都可见一斑。从企业的长远发展来看，这种倾向性是绝对正确且必要的。但从纯粹的技术生态角度而言，商业化是一把双刃剑，一方面它会带来更多的资本投入，有利于促进产品的发展，另一方面也意味着一个封闭的生态壁垒正在形成。

在开源生态中，Fluentd 是个十分不错的备选替代品。作为 CNCF 基金会项目之一，它在第一时间得到了 Docker 和 Kubernetes 等容器平台的优先支持，例如在 Docker 原生内置的日志输出驱动中，Fluentd 就位列其中^①。

^① <https://docs.docker.com/engine/admin/logging/fluentd/>

Fluentd 集合了 Logstash 的强大功能和 Filebeat 的轻量部署,可以看作两者的折中体现。从性能上看,Fluentd 的空载内存使用约 40MB,高于 Filebeat(约 13MB),但远低于 Logstash(接近 600MB),空载时的 CPU 使用低至可以忽略不计(即使在单核 CPU 上占用率都小于 0.1%),运行时的 CPU 消耗略高,但同样介于 Filebeat 与 Logstash 之间。在功能方面,Fluentd 通过插件组合能形成复杂的链式匹配处理规则,以及自动聚合在一段时间内反复出现的相同日志内容的能力,并能将复杂的配置规则外置到多个独立的规则文件中。与 Logstash 功能相当,而远远比 Filebeat 强大。表 7-1 展示了 Fluentd 与 Filebeat、Logstash 的详细差异对比。

表 7-1 Fluentd 与 Filebeat、Logstash 的对比

	功 能	性 能	部 署	配 置
Fluentd	可通过插件扩展,数据源、输出目标和过滤器都比较丰富	底层核心使用 C++实现,数据处理逻辑和插件使用 Ruby 语言,性能较好	需要 Ruby 运行环境,安装插件需要网络和 C++编译环境,较为烦琐。不过若使用容器部署则十分简单	专用 DSL 格式,配置灵活,可拆分多个子文件管理,可实现复杂判断逻辑
Filebeat	不可扩展,内置常用的数据源和输出目标,能对日志内容进行过滤和处理	整体使用 Golang 实现,性能优秀	单个二进制文件,部署十分简单,也可使用容器方式部署	基本配置采用 YAML 格式,容易阅读,但定义复杂处理过程需要使用模块扩展,较为烦琐
Logstash	可通过插件扩展,数据源、输出目标和过滤器都比较丰富	整体使用 JRuby 实现,资源消耗较多,性能一般,通常需要使用消息队列或前置 Redis 缓存等机制来规避一些性能问题	需要 Java 运行环境,单机部署比较容易,但以集群方式部署时涉及外置缓存或消息队列等额外配置,即使使用容器部署依然较为烦琐	类似 JSON 的 DSL 格式,可拆分多个子文件管理,可实现复杂判断逻辑

从关注点分离的设计上来说,Filebeat+Logstash 在性能 / 功能的取舍和分工方面似乎更加合理,但从实际使用的角度来看,Fluentd 这种两头兼顾的策略在小规模集群里的确能简化一些部署和集成的麻烦。

Fluentd 的官方镜像中仅仅内置了核心的插件,通常在使用前需要先制作包含所需插件的服务镜像。下面这个 Dockerfile 可以作为此类镜像的模板。

```
FROM fluent/fluentd:v0.14.17
RUN apk --no-cache add sudo build-base ruby-dev && \
    gem install --no-rdoc --no-ri \
```

```
fluent-plugin-kafka \  
fluent-plugin-grepcounter \  
fluent-plugin-elasticsearch \  
fluent-plugin-record-reformer && \  
gem sources -c && \  
apk del sudo build-base ruby-dev
```

作为例子，这个 Dockerfile 在镜像里添加了输出到 Kafka 和 Elasticsearch 的插件，以及用于关键词计数和重写标签的插件，使用以下命令将这个镜像构建出来。

```
$ docker build -t fluentd-with-plugin:v0.14.17 ./
```

在实际使用的时候，通常还会把服务的配置一起打入到镜像中，以实现即启即用，在需要更新配置的时候通过部署流水线自动重新打包然后替换旧容器。但这个例子尽量简化场景，依然使用从外部挂载配置文件的方式。下面的配置会让 Fluentd 采集 “/log” 目录（启动容器时将系统的 “/var/log” 目录挂载到这个位置）和 “/container” 目录下的所有日志文件（启动容器时将 Docker 安装的根目录挂载到这个位置），然后将结果以 Logstash 的格式写入到 Elasticsearch 服务（以 Logstash 格式写入部署只是为了更符合主流用户习惯）。为了让每个配置文件不至于太过庞大复杂，这里利用 Fluentd 的 “@include” 语法将配置分成多个文件。先创建两个分别用于采集容器和系统日志的配置文件，如下所示。

```
$ sudo mkdir -p /etc/fluentd/subconf/  
  
$ cat <<EOF | sudo tee /etc/fluentd/subconf/docker.conf  
<source>  
  @type tail  
  path /containers/*/*-json.log  
  pos_file /var/log/fluentd-docker.pos  
  time_format %Y-%m-%dT%H:%M:%S  
  tag docker.*  
  format json  
</source>  
<filter docker.**>  
  @type grep  
  regexp1 log \[(ERROR|WARN)\]  
</filter>  
<match docker.var.lib.docker.containers.*.*.log>  
  @type record_reformer  
  container_id ${tag_parts[5]}  
  tag docker.all  
</match>  
<match docker.all>
```

```

@type elasticsearch
host 172.31.31.164
logstash_format true
flush_interval 5s
include_tag_key true
tag_key docker
</match>
EOF

$ cat <<EOF | sudo tee /etc/fluentd/subconf/file.conf
<source>
  @type tail
  format none
  path /log/demo/*.log
  pos_file /var/log/fluentd-demo.pos
  tag file.*
</source>
<filter file.**>
  @type grep
  regexp1 message \[(ERROR|WARN)\]
</filter>
<match file.**>
  @type elasticsearch
  host 172.31.31.164
  logstash_format true
  flush_interval 5s
  include_tag_key true
  tag_key file
</match>
EOF

```

再创建一个作为入口的配置文件，引用所有其他配置文件，如下所示。

```

$ cat <<EOF | sudo tee /etc/fluentd/fluent.conf
@include subconf/*.conf
EOF

```

关于 Fluentd 配置文件的详细写法，可以参考它的文档^①。

在启动 Fluentd 容器启动时，默认会读取“/fluentd/etc/fluentd.conf”文件作为服务的配置，注意以下命令中的目录映射关系。同时，为了让采集到的日志标签中的主机名称与实际主机名称一致，可以把节点的主机名传入容器中。

① <http://docs.fluentd.org/v0.14/articles/config-file>

```
$ docker run -d --name fluentd \
  -p 24224:24224 \
  -h $(hostname) \
  -v /etc/fluentd:/fluentd/etc/ \
  -v /var/log:/log:ro \
  -v /var/lib/docker/containers:/containers:ro \
  fluentd-with-plugin:v0.14.17
```

由于 Fluentd 的运行性能较好，可以将一部分数据计算的工作放在各采集节点完成，但有时为了减少对采集节点的资源占用以及对数据进行跨节点统计，依然会使用 Fluentd 级联的形式，即各采集节点只负责将采集到的数据汇聚到统一的 Fluentd 集群里，然后在这些节点进行数据的处理计算。例如 `grepcounter` 插件可以统计在一段时间内，所有收集到的特定标签数据里指定内容出现的次数，这类统计数据放在级联的上层 Fluentd 节点就比较合适。Fluentd 插件的配置大都简明易读，下面是一个 `grepcounter` 插件使用的例子。

```
<match log.service-name.**>
  @type grepcounter
  count_interval 10
  input_key message
  regexp (\[ERROR\]|\[FAILED\])
  threshold 1
  add_tag_prefix error.service-name
</match>
```

作为高可用的部署方案，通常建议在汇聚节点放置至少一套冗余的 Fluentd 服务，以避免汇聚节点出现单点故障，这种部署结构如图 7-16 所示。

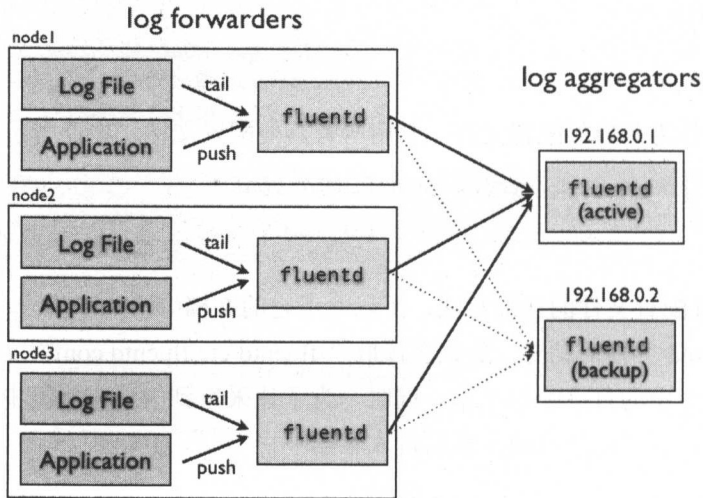


图 7-16 级联的 Fluentd

采集到 Fluentd 里的日志汇总到 Elasticsearch 以后的检索和调试方法与前一小节介绍的基本一致，这里不再重复讲解。

除了将日志存入 Elasticsearch 进行检索，对于数量比较大的日志，也可以让 Filebeats 或 Fluentd 直接写入 Kafka 队列，然后在队列的另一端通过 Storm 或 Spark 等流数据处理程序进行实时的日志分析，或接入 Prometheus 中，根据日志中关键字频率等信息产生必要的告警。关于汇聚后的日志数据利用同样是运维领域中十分值得探讨的话题，这部分内容超出了本书的范围，有兴趣的读者可以在网络上搜索“日志分析”等关键词以找到更多相关内容。

7.3 服务发现

7.3.1 常见的服务发现方案

“服务发现”原本是通信领域中的一个名词，用于表示多个网络设备之间通过唯一的 UUID 标识和专用的协议自动发现对方的方式，例如蓝牙的配对就是典型的服务发现运用。那么容器集群中为什么也需要服务发现呢？

在集群中部署的许多服务之间经常要相互通信，但由于集群本身的动态特性，每个服务的后端容器数量、使用的 IP 地址和对应的端口都有可能随时变化。因此如何管理各个服务当前运行的地址，并让它们能方便可靠地被获取，就成为集群环境必须解决的一个问题。

服务发现的具体实现可以分为好几种，比如环境变量注入、DNS 记录或者是专用的服务注册中心。

环境变量注入的典型场景是早期 Docker 的 `--link` 参数，通过这种方法关联的容器会自动在运行上下文中注入一些额外的环境变量，环境变量通常会以服务名称、协议和原始端口命名，比如“`SERVICE_A_TCP_PORT_8000`”，容器中的服务只需读取环境变量就可以获得要连接的目标地址和端口。环境变量的方法简单易用，但一旦被依赖服务地址发生变化，必须重启所有依赖它的服务容器才能让新地址生效，因此通常只适用于服务运行地址不会变化或每次更新都进行整体部署的小型系统。

虽然普通的 DNS 记录保存的仅仅是域名和 IP 地址之间的转换关系，在 DNS 协议标准中有一种专门用于保存服务信息的“SRV 记录”，这种记录能够包含服务名称到域名（或

IP 地址)和端口的信息。只要每次服务启动后(由平台或服务自己)将新的监听 IP 和端口信息通过 SRV 记录更新到 DNS, 其他服务就能以 DNS 标准协议解析出该服务的运行地址了。不过由于 DNS 协议允许多级缓存, 每次服务地址变化可能需要几分钟甚至几十分钟的时间才会在整个集群中被识别到, 这种方式并没有得到广泛的使用。

既然标准的 DNS 协议不靠谱, 不妨模拟这种机制定义一个无缓存的服务信息存储器, 这就是专用的服务注册中心。它提供两个最基本的操作, 如下所示。

- 服务注册: 让服务启动时将自己的地址和端口注册到集中存储中心。
- 服务发现: 让其他服务可以查找已注册的服务位置信息。

这种特殊的记录存储服务, 首先需要处理服务故障退出时的记录注销问题, 像 MySQL、MongoDB 等普通数据库服务由于不支持数据的 TTL (Time To Live) 特性, 若因服务意外退出而没有将自己从注册中心正常注销, 就会导致请求者得到错误的数据, 因此不适合作为服务注册中心的存储。主流的缓存服务如 Redis、Memcache 能够支持数据 TTL, 但它们通常使用未加密的传输协议以及 TCP 长连接进行数据传输。一些业务系统中的服务可能会散布在多个机房中不同集群, 甚至中间通过 Internet 公共网络连接, 最好采用能够加密且适合远程通信的协议(如 HTTP), 因此这些服务也不太适合作为注册中心的存储。此外, 还应该考虑单点故障、网络分区和其他分布式环境下可能出现的高可用和一致性问题。Paxos 选举协议是第一个真正被证明能解决分布式一致性问题的协商协议, Raft 协议是它的简化版本, 第 2 章简单介绍过 Raft 协议随机选择主节点的投票机制。目前实现了 Paxos 协议的开源存储服务主要有 ZooKeeper 和 Doozer, 而实现 Raft 协议的开源存储项目则有 Etcd 和 Consul。

不论这些协议表面上如何复杂(这是一致性选举的协议, 为了确保数据一致, 对存储本身来说没什么关系), 不难看出, 服务注册中心本质上还是一个存储键值数据的特殊数据库, 因此这些产品通常也可以在分布式系统中用于存储服务的配置数据和其他运行数据。

ZooKeeper 是这些项目中历史最悠久的, 它属于 Apache Hadoop 项目集中的子项目, 采用 Java 编写。ZooKeeper 曾经一度是这个领域中的明星产品, 以至于虽然存在着资源消耗大、(部署简单但是)维护麻烦等许多问题, 在许多项目中依然广泛使用, CoreOS 还专门推出了一个将 Etcd 适配成 ZooKeeper 协议的 Golang 轻量级代理: Zetcd^①, 以吸引那些还在使用 ZooKeeper 的服务逐渐转向 Etcd。

Doozer 是使用 Golang 编写的一款轻量级 Paxos 协议存储服务。意图改变 ZooKeeper

^① <https://github.com/coreos/zetcd>

的笨重和复杂情况，不过这个项目目前已经没有人在维护了，因此也逐渐退出了大众的视野。

Etcd 是 CoreOS 公司推出的强一致性存储服务，采用 Go 语言编写，受到 ZooKeeper 和 Doozer 的启发，通过 Raft 协议来保证数据一致。目前有 v2 和 v3 两个互不兼容的设计模型，采用两种完全不同的底层存储和通信协议，后者提供了更多的功能以及更高的读写效率，但由于早期 Etcd 已经积累的较大的用户基数，因此两个模型的实现将继续在未来很长的一段时间里同时存在。

Consul 是 Hashicorp 公司的企业级服务注册中心产品，设计之初就是为了作为专用的服务发现平台，内置了 DNS 域名服务以及服务健康监控等附加功能，却在前期的很长一段时间里都没有提供命令行操作键值存储的命令（只有主要用于调试数据的键值存储 API，直到 v0.7 版本以后才加入直接操作存储的 `consul kv` 命令）。Consul 的 Gossip 协议非常适合用于充满不确定性因素的集群网络环境，它甚至充分考虑了跨数据中心的服务通信的应用场景，并为此设计了“WAN Gossip”成员池的逻辑抽象。

下面将重点介绍 Etcd 和 Consul，它们是当下最主流的强一致性分布式键值存储产品，也代表了目前服务注册中心的发展趋势。

7.3.2 Etcd

Etcd 最初的定位是高性能的通用键值存储服务，不过随着开源社区捣鼓出各种编程语言的 Etcd SDK，很多需要服务注册发现的场景都能够通过 Etcd 轻松实现。

Etcd 从其 3.0 版本开始同时提供两套 API，分别对应 v2 和 v3 两种的存储模型，调用不同 API 存储的数据是独立的，相互不可见。其中 v3 模型使用 gRPC 协议通信（同时提供一个 HTTP/HTTPS 协议的 API 代理），支持单节点百万级键值数据存储，支持数据监控和 TTL 租约的聚合管理，在性能上明显优于 v2 模型。因此，下面仅介绍 v3 模型的命令和 API 使用。

Etcd 服务的部署比较简单，首先从 GitHub 下载最新的 Etcd 二进制包，如下所示。

```
$ wget https://github.com/coreos/etcd/releases/download/v3.2.0/etcd-v3.2.0-linux-amd64.tar.gz
```

解压后，将程序目录中的“etcd”和“etcdctl”可执行文件拷贝到系统 PATH 变量中的任意一个目录，例如“/usr/local/bin/”，如下所示。

```
$ tar xzf etcd-v3.2.0-linux-amd64.tar.gz
$ sudo cp etcd-v3.2.0-linux-amd64/etcd* /usr/local/bin/
```

其中“etcd”是服务的后台进程，“etcdctl”是用于与 Etcd 服务交互的命令行工具。根据不同的使用场景，Etcd 有几种典型的部署模式，如下所示。

- 本地模式。
- 单点模式。
- 集群模式。

不带任何参数启动的 etcd 服务会进入本地模式。这种模式下其实是 Etcd 单点模式的特殊场景，此时它默认监听本地网卡 127.0.0.1 地址的 2379 端口用于提供 API 以及与客户端通信，由于仅仅监听本地 IP，只有在相同节点上的服务才能连接到它。因此本地模式通常只是用于依赖 Etcd 的服务在开发和测试阶段快速启动 Etcd 进行功能验证，无法真正对外提供服务。

```
$ etcd
```

Etcd 默认运行在前台，建议将 Systemd 作为系统服务，或制作成容器镜像通过 Docker（或其他容器服务）运行。

启动时带有 `--force-new-cluster` 参数的 Etcd 服务会进入单点模式，如下所示。在这个模式下，Etcd 服务可以绑定到任何指定的 IP 地址（包括 0.0.0.0，表示所在节点的全部可用 IP 地址）。需要注意的是，一旦 Etcd 服务以单点模式启动，它是不能在不丢失数据的情况下，通过添加其他成员或加入其他集群改变为集群模式的。因此，在产品环境以及未来可能需要扩展的环境里，请妥善考虑避免使用单点模式启动 Etcd 服务。

```
$ etcd --name infra0 \  
--listen-client-urls http://0.0.0.0:2379 \  
--listen-peer-urls http://0.0.0.0:2380 \  
--advertise-client-urls http://10.30.2.20:2379 \  
--initial-advertise-peer-urls http://10.30.2.20:2380 \  
--initial-cluster infra0=http://10.30.2.20:2380 \  
--initial-cluster-token etcd-cluster-0 \  
--initial-cluster-state new \  
--force-new-cluster
```

这个命令需要提供的参数较多，解释如下。

- **name:** 节点在 `etcdctl member` 命令中展示的节点名称，可以是任意字符串。
- **listen-client-urls:** 提供 API 和客户端连接的端口，地址可以是 0.0.0.0，通常使用 2379 端口。
- **listen-peer-urls:** 与其他 Etcd 节点进行通信的端口，地址可以是 0.0.0.0，通常使用 2380 端口。

- **advertise-client-urls**: 提供给客户端作为回调的服务连接地址，必须是具体 IP，通常使用 2379 端口。
- **initial-advertise-peer-urls**: 提供给其他 Etcd 节点作为回调的服务连接地址，必须是具体 IP，通常使用 2380 端口。
- **initial-cluster**: 初始集群成员，使用“节点名称=节点 Peer 地址”的格式，对于单节点模式来说，即表示上一个参数的地址。
- **initial-cluster-token**: 用于在同一个网络里存在多个 Etcd 集群时，区分各集群数据的一个集群标识。
- **initial-cluster-state**: 若是创建新的空集群则该值应使用 **new**，若是在已经有数据的集群加入新成员则该值应使用 **existing**。
- **force-new-cluster**: 强制进入单节点模式。

集群模式是 Etcd 在正式场合中使用的推荐模式，具体的部署方法又可以细分成“静态部署”和“基于服务发现部署”。静态部署集群的启动方式与单点模式很像，只是在 **-initial-cluster** 参数里需要列出所有集群的成员，且不必添加 **--force-new-cluster** 参数，如下所示。

```
# 节点 01
$ etcd --name infra1 \
  --listen-client-urls http://0.0.0.0:2379 \
  --listen-peer-urls http://0.0.0.0:2380 \
  --advertise-client-urls http://10.30.2.21:2379 \
  --initial-advertise-peer-urls http://10.30.2.21:2380 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster 'infra1=http://10.30.2.21:12380,
    infra2=http://10.30.2.22:22380,infra3=http://10.30.2.23:32380' \
  --initial-cluster-state new

# 节点 02
$ etcd --name infra2 \
  --listen-client-urls http://0.0.0.0:2379 \
  --listen-peer-urls http://0.0.0.0:2380 \
  --advertise-client-urls http://10.30.2.22:2379 \
  --initial-advertise-peer-urls http://10.30.2.22:2380 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster 'infra1=http://10.30.2.21:12380,
    infra2=http://10.30.2.22:22380,infra3=http://10.30.2.23:32380' \
  --initial-cluster-state new

# 节点 03
```



```
$ etcd --name infra3 \  
  --listen-client-urls http://0.0.0.0:2379 \  
  --listen-peer-urls http://0.0.0.0:2380 \  
  --advertise-client-urls http://10.30.2.23:2379 \  
  --initial-advertise-peer-urls http://10.30.2.23:2380 \  
  --initial-cluster-token etcd-cluster-1 \  
  --initial-cluster 'infra1=http://10.30.2.21:12380,  
                    infra2=http://10.30.2.22:22380,infra3=http://10.30.2.23:32380' \  
  --initial-cluster-state new
```

根据 Raft 协议要求，为了避免在选举投票时出现平局的情况，集群的节点总数应该为单数，通常使用 3 个或者 5 个 Etcd 节点构成集群。只要还有半数以上的节点存活，这个集群就能够继续正常工作，关于 Raft 选举 Leader 节点的过程，在第 2 章中已经介绍过。另外，在相同 Etcd 集群中的节点应该保持 `--initial-cluster-token` 参数值一致，否则将无法正常通信。部署完成后，在其中任意一个节点上的数据操作都会自动同步到整个集群上。

基于服务发现部署的 Etcd 集群可以从一个外部系统（另一个 Etcd 服务或 DNS 记录）自动获得集群成员信息，这种场景主要用在 CoreOS 系统的自动集群组建上。对于作为服务注册中心的 Etcd，倘若使用另一个服务发现机制来组建集群，就会陷入鸡生蛋、蛋生鸡的悖论，读者若对这种组件集群的方式有兴趣，不妨阅读它的配置文档^①。

为了简便起见，上述部署过程没有包含开启 TLS 加密传输，若希望使用通信加密，在启动 Etcd 服务时还需添加 `--ca-file`、`--cert-file`、`--key-file`、`--peer-ca-file`、`--peer-cert-file`、`--peer-key-file` 参数来分别指定与客户端、其他 Etcd 节点之间的根证书、通信加密证书和密钥文件。

Etcd 的 v3 模型采用基于 Protobuf 的 gRPC 协议，常规的 `curl` 工具无法直接调用它的接口，需使用 Etcd 安装包中内置的 `etcdctl` 命令行工具，或是特定编程语言的 SDK 来进行操作。为了最大限度地保持兼容性，`etcdctl` 命令默认使用 v2 模型的 API 与 Etcd 服务通信。若要启动 v3 模型，需要在运行该命令的上下文环境中将“ETCDCTL_API”环境变量的值设置为“3”。注意观察添加此变量前后 `etcdctl` 命令输出的版本信息差异，如下所示。

```
$ etcdctl --version  
etcdctl version: 3.2.0  
API version: 2  
  
$ export ETCDCTL_API=3  
$ etcdctl version
```

^① <https://github.com/coreos/etcd/blob/master/Documentation/op-guide/clustering.md>

```
etcdctl version: 3.2.0
API version: 3.2
```

此小节后面的部分均假设已经在上下文中设置过 `ETCDCTL_API` 环境变量。Etcd 最基本的操作是添加、查询、修改和删除键值数据，添加和删除可以用 `etcdctl put` 和 `etcdctl get` 命令实现，如下所示。

```
$ etcdctl put demo "hello container"
OK
$ etcdctl get demo
demo
hello container
```

在 Etcd 中存储的所有键和值都是字符串。除了单个读取键的内容，`etcdctl get` 命令可以一次性获取指定两个键之间按字符串比较所得的所有中间部分键的值（取值范围为前闭后开区间），如下所示。

```
$ etcdctl put demo1 "hello etcd"
$ etcdctl put demo2 "hello consul"
$ etcdctl put demo3 "hello zookeeper"
$ etcdctl get demo1 demo4 --print-value-only
hello etcd
hello consul
hello zookeeper
```

也可以批量获得匹配特定字符串开头的键的所有键的内容，如下所示。

```
$ etcdctl get --prefix demo --print-value-only
hello container
hello etcd
hello consul
hello zookeeper
```

键的更新同样使用 `etcdctl put` 命令，如下所示。此外，Etcd 还支持读取指定键的任意一个历史版本的值，这是 Etcd 服务的一个重要特性。

```
$ etcdctl put demo "hello swarmkit"
$ etcdctl put demo "hello kubernetes"
$ etcdctl put demo "hello mesos"
$ etcdctl put demo "hello rancher"
$ etcdctl get demo -w json | jq .
{
  "header": {
    ...
```

```
    },
    "kvs": [
      {
        "key": "ZGVtbw==",
        "create_revision": 2,
        "mod_revision": 10,
        "version": 5,
        "value": "aGVsbG8gcmFuY2hlcg=="
      }
    ],
    "count": 1
  }
}
```

解释一下在 `kvs` 中各属性的含义，如下所示。

- **key**: Base64 编码过的键（“ZGVtbw==” 解码后即为 `demo`）。
- **create_revision**: 该键是在哪个版本时创建的。
- **mod_revision**: 当前值是在哪个版本时写入的。
- **version**: 该键一共更新过几个版本。
- **value**: Base64 编码过的值。

需要注意的是，这里的“版本”并不是每个键单独计数的，而是针对整个 Etcd 系统，在 Etcd 中的任意修改都会使得这个版本号增加。因此上述信息表示的是，所查询的这个键在系统序号为 2 的变更中被创建，当前的值是由系统序号为 10 的那次变更写入的，该键一共被更新过 5 次。在查询时使用 `--rev` 参数指定一个序号，就可以看到在特定的时间上这个键的内容是什么，如下所示。

```
$ etcdctl get demo --rev 10
hello rancher
$ etcdctl get demo --rev 9
hello mesos
$ etcdctl get demo --rev 8
hello kubernetes
$ etcdctl get demo --rev 7
hello swarmkit
```

默认情况下 Etcd 始终保存所有修改历史，时间长了后可能会占用不必要的磁盘空间。可以在启动“Etcd”服务时添加 `--auto-compaction-retention` 参数指定自动删除历史的时间，或使用 `etcdctl compaction` 来手动删除多余的历史记录。例如删除版本 4 以前的所有记录信息，如下所示。

```
$ etcdctl compaction 4
```

删除指定的键可以使用 `etcdctl del` 命令，如下所示。注意，只要没有执行过历史版本清理的操作，被删除键的内容依然可以在历史版本中被查询到。

```
$ etcdctl del demo
```

Etcd 支持原子性的“读—判断—写”操作，确保只有当指定的键存在或为特定值时才执行操作，如下所示。

```
$ cat <<EOF | etcdctl txn
val("demo") = "hello"

put res "ok"

put res "fail"

EOF
```

这个命令的内容分为三段，第一段为判断条件，即键 `demo` 的值为 `hello`，除了用来判断值的 `value`，还可以用 `mod` 操作获得键的状态，如用 `mod("stuq") > "0"` 作为条件来判断指定的键是否存在。第二段为若判断成功时执行的操作，即将 `res` 键写入 `ok`。第三段为判断失败时执行的操作，即将 `res` 键写入 `fail`。段与段之间使用空行分隔，每段可以写入多个操作，Etcd 确保整个命令的执行都以原子操作的形式一次性下发的。

使用 `etcdctl watch` 可以监控指定键的变化，并且将监听到的创建、修改、删除持续地打印出来，如下所示。

```
$ etcdctl watch demo
```

监听操作默认是从命令发起时开始的，也可以指定从任意一个历史版本的时间点开始，如下所示。

```
$ etcdctl watch --rev=8 demo
```

在 Etcd 中对写入的数据可以有 TTL (Time to live) 的支持，即在一段时间后自动将数据删除，这有点像缓存的功能。Etcd 的 v3 模型对 TTL 的支持与 v2 模型有比较大的差异，特别是引入了“租约”的概念，每个租约会被关联到一组键上，共享相同的 TTL 时间。例如，创建一个回收时间为 600s 的租约，如下所示。

```
$ etcdctl lease grant 600
lease 694d5cd54aa0dd24 granted with TTL(600s)
```

创建一个与该租约关联的数据，如下所示。


```
$ etcdctl put --lease 694d5cd54aa0dd24 demo "hello container"
OK
```

查看租约的剩余时间，如下所示。

```
$ etcdctl lease timetolive --keys 694d5cd54aa0dd24
lease 694d5cd54aa0dd24 granted with TTL(600s), remaining(458s), attached
keys([demo])
```

在这个时间租约到期或被人为收回之前，可以通过 `etcdctl lease keep-alive` 续约，即重设租约时间，如下所示。

```
$ etcdctl lease keep-alive 694d5cd54aa0dd24
```

这个命令会保持不停刷新租约的起始时间，确保租约不到期，可用“Ctrl+C”来停止它。在租约到期前，可以被人为提前回收。不论租约是超时还是被人为回收，都会使得所有关联在该租约上的数据被删除，如下所示。

```
$ etcdctl lease revoke 694d5cd54aa0dd24
lease 694d5cd54aa0dd24 revoked
```

Etcd 集群的成员同样，可以用 API 进行管理，相应的命令是 `etcdctl member`。例如查看当前集群中的成员，如下所示。

```
$ etcdctl member list
8211f1d0f64f3269, started, infra1, http://10.30.2.21:2380, http://10.30.2.21:2379
91bc3c398fb3c146, started, infra2, http://10.30.2.22:2380, http://10.30.2.22:2379
fd422379fda50e48, started, infra3, http://10.30.2.23:2380, http://10.30.2.23:2379
```

添加一个新的成员到集群需要两步操作，首先在现有的集群里通过 `etcdctl member add` 声明新节点的加入，然后使用正确的参数启动这个节点，如下所示。

```
$ etcdctl member add infra4 --peer-urls http://10.30.2.24:2380

$ etcd --name infra4 \
  --listen-client-urls http://0.0.0.0:42379 \
  --listen-peer-urls http://0.0.0.0:2380 \
  --advertise-client-urls http://10.30.2.24:2379 \
  --initial-advertise-peer-urls http://10.30.2.24:42380 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-cluster 'infra1=http://10.30.2.21:12380,
                    infra2=http://10.30.2.22:22380,infra3=http://10.30.2.23:32380,
                    infra4=http://10.30.2.24:42380' \
  --initial-cluster-state existing
```


采用两步加入不仅可以防止外部节点未经授权私自加入集群窃取数据，还能提前检查节点参数的错误，例如使用了重复的 ID 号等。删除节点则直接在集群中执行 `etcdctl member remove` 指定节点 ID 即可，如下所示。

```
$ etcdctl member remove ef63616b23f8c8c4
```

Etcd 数据的备份和还原由 `etcdctl snapshot` 命令完成。例如创建一个数据备份，如下所示。

```
$ etcdctl snapshot save snapshot.db
Snapshot saved at snapshot.db
```

一旦集群遇到无法恢复的严重事故时（例如半数以上节点丢失），可以使用备份文件重建集群。例如在一个新节点上恢复备份的内容，如下所示。

```
# 还原数据
$ etcdctl snapshot restore snapshot.db \
  --initial-cluster-token etcd-cluster-1 \
  --initial-advertise-peer-urls http://10.30.4.31:2380 \
  --name sshot1 \
  --initial-cluster 'sshot1=http://10.30.4.31:2380,
    sshot2=http://10.30.4.32:2380,sshot3=http://10.30.4.33:2380'

# 重建集群
$ etcd --name sshot1 \
  --listen-client-urls http://0.0.0.0:2379 \
  --advertise-client-urls http://10.30.4.31:2379 \
  --listen-peer-urls http://0.0.0.0:2380
```

然后在另外两个新节点上重复以上命令（IP 地址相应调整），这样就得到了一个内容与之前备份的集群完全相同的新集群。

除了使用 TLS 证书实现加密传输，Etcd 也支持基于用户角色授权的访问控制机制，相关的命令是 `etcdctl user` 和 `etcdctl role`。

在开启授权验证前，需要为 Etcd 的默认管理员账号 `root` 设置密码，设置后请务必记住 `root` 用户的密码，否则将无法关闭授权功能。`root` 用户会被默认加入同名的 `root` 角色中，该角色具有所有键值的读写权限，如下所示。

```
$ etcdctl user add root
New password: ***
Type password of root again for confirmation: ***
User root created
```

可以创建更多的普通用户和角色，如下所示。角色用于配置对特定键的读写权限，用户与角色关联后用于操作者身份验证。

```
$ etcdctl user add demoUser
$ etcdctl role add demoRole
```

将用户赋予角色。然后为角色赋予指定的权限：可以读写名为“demo”的键，且可以读取任何其他以 demo 开头的键，如下所示。

```
$ etcdctl user grant-role demoUser demoRole
Role demoRole is granted to user demoUser

$ etcdctl role grant-permission demoRole readwrite demo
Role demoRole updated
$ etcdctl role grant-permission demoRole read --prefix demo
Role demoRole updated
```

现在开启 Etcd 的访问授权验证，如下所示。

```
$ etcdctl auth enable
Authentication Enabled
```

一旦开启授权，所有直接对 Etcd 服务的读写都会被拒绝。若使用 etcdctl 命令，必须使用 --user 参数指定操作者，如下所示。

```
$ etcdctl get demo
Error: etcdserver: user name is empty

$ etcdctl --user demoUser get demo
Password: ***
demo
hello container
```

只有符合权限控制的请求才会被执行，如下所示。

```
$ etcdctl --user demoUser put demo1 "hello etcd"
Password: ***
Error: etcdserver: permission denied
$ etcdctl --user demoUser get demo1
Password: ***
demo1
hello swarmkit
```

只有 root 用户可以关闭授权功能，如下所示。

```
$ etcdctl --user root auth disable
```

```
Password: ***
Authentication Disabled
```

etcdctl 还提供了进行远程数据同步的 `make-mirror`、分布式 Mutex 锁命令 `lock`、v2 模型到 v3 模型数据迁移的命令 `migrate` 等，这里不再逐一介绍。

使用过 Etcd v2 模型的用户可能知道 Etcd 的 Proxy 功能，它会启动一个自身不保存任何数据的 Etcd 代理节点，所有访问该节点的请求都会被转发到真实的 Etcd 集群上。相当于 Etcd 的一层反向代理，能够确保用户始终访问健康的 Etcd 节点（避免用户直接与 Etcd 集群中的某个具体节点连接，在系统中留下单点故障隐患）。在 Etcd 的 v3 模型中重新设计了 Proxy 的功能，使它不但能够作为访问代理，还能聚合所有连接在代理节点上的 `watch` 操作，将大量的监控请求其合并，从而极大地降低 Etcd 节点的服务压力。使用 `etcd grpc-proxy start` 命令可以启动一个 v3 模式的代理节点，如下所示。

```
$ etcd grpc-proxy start --listen-addr 0.0.0.0:2379 \
  --endpoints 10.30.2.21:2379,10.30.2.22:2379,10.30.2.23:2379
```

除了用于反向代理的 Proxy，Etcd 还将提供了将 gRPC 接口转换成其他协议的代理服务，例如上个小节中提到的 ZooKeeper 协议代理 Zetcd，还有 Etcd 内置的一个 HTTP 协议代理，用于支持没有 gRPC 实现的语言，如下所示。

```
$ etcd gateway start --listen-addr 127.0.0.1:9000 \
  --endpoints 10.30.2.21:2379,10.30.2.22:2379,10.30.2.23:2379
```

这个代理功能目前还在 Alpha 阶段，官方已经提供了 API 列表的 Swagger 文档^①，使用举例如下所示。

```
$ printf demo | base64
ZGVtbw==
$ printf "hello etcd" | base64
aGVsbG8gZXRjZA==

$ curl -L http://localhost:9000/v3alpha/kv/put -X POST \
  -d '{"key": "ZGVtbw==", "value": "aGVsbG8gZXRjZA=="}'

$ curl -L http://localhost:9000/v3alpha/kv/range -X POST -d '{"key": "ZGVtbw=="}'
```

单纯的 Etcd 仅提供了一种高可靠的分布式数据存储能力，如果将它用于服务发现，还需要将服务的信息注册到这个存储服务中，具体的实现方式主要有两种。第一种方法是

① <https://github.com/coreos/etcd/blob/master/Documentation/dev-guide/apispec/swagger/rpc.swagger.json>

让每个运行的服务启动时自己写入运行信息到 Etcd 里，这种方法的可定制性比较强，但也具有较强的侵入性，需要对现有项目进行功能改造。第二种方法是用一个作为第三者的服务，通过监听容器（如 Docker）的事件，自动把识别到的服务写到 Etcd 存储里，开源项目 Registrator^①可以方便地完成这个工作。Registrator（原名 Docksul）能够监听指定远端主机上的 Docker 运行容器的启动和停止事件，并自动将容器的地址和状态同步注册到 Etcd 里（也支持注册到 Consul），这样便确保了 Etcd 中的服务记录数据与集群中所有运行在容器的服务保持一致。

7.3.3 Consul

相比 Etcd 的稳定、简单，Consul 则更加实用，提供了诸如域名解析、健康检查以及专用的服务注册发现 API，在普通键值存储方面也不逊于 Etcd。表 7-2 详细对比了它们之间的差异。

表 7-2 Etcd 与 Consul 的对比

	Etcd v2	Etcd v3	Consul
协议	HTTP/HTTPS	gRPC	HTTP/HTTPS
分布式键值存储	√	√	√
TTL 租约	√	√	√
监控变化	√	√	√
加密通信&用户权限	√	√	√
事务（原子读写）	√	√	√
服务注册发现 ^②	×	×	√
健康检查	×	×	√
域名服务	×	×	√
多数据中心	×	×	√
分布式锁	×	√	√
历史记录	√	√	×

可以看出，Consul 的功能比 Etcd 丰富得多，但唯独缺少了数据的历史记录，这与两者的设计定位有关。Etcd 最初的设计主要用于配置信息的键值存储，对于配置信息而言，可

① <https://github.com/gliderlabs/registrator>

② 这里指的是有专用的服务注册和发现 API，Etcd 的键值 API 同样可以用于服务注册发现，并且是一种很常见的用法。

追溯变更历史是一个十分有用的加分项。而 Consul 是专为服务发现的场景设计的，键值存储仅仅算是附带提供的功能。

Consul 在 GitHub 页面上只提供各版本源码的下载链接，预编译的二进制文件可以在它的官方网站下载到^①，如下所示。

```
$ wget https://releases.hashicorp.com/consul/0.8.4/consul_0.8.4_linux_amd64.zip
$ unzip consul_0.8.4_linux_amd64.zip
```

解压后只有一个二进制文件，既是命令行工具，也可以用于启动服务。

启动服务的命令是 `consul agent`，根据服务运行的方式，同样可以分成本地模式、单节点模式和集群模式。本地模式的服务的创建比较简单，启动时使用 `-dev` 参数即可。同时为了方便进行服务注册，建议加上 `-config-dir` 参数指定一个配置目录，如下所示。

```
$ sudo mkdir /etc/consul.d && sudo chown `whoami` /etc/consul.d
$ consul agent -dev -config-dir=/etc/consul.d
```

与 Etcd 一样，本地模式的 Consul 监听 127.0.0.1 地址，只有本地运行的服务可以连接，通常用于开发和本地调试。Consul 将节点分为 Client 和 Server 两种角色，只有 Server 角色的节点会参与 Leader 选举，Client 节点相当于 Etcd 中的 Proxy 角色。默认情况下，Consul 启动的节点都会以 Client 角色运行，可以通过 `-server` 参数将节点指定为 Server 角色。使用 `-dev` 参数启动的节点其实也是一种特殊的 Server 节点。

在 Consul 官方文档中其实是没有“单节点模式”与“集群模式”这两种说法的。本书有意将使用只有一个 Server 节点的 Consul 集群称为“单节点模式”的 Consul，因为在这种情况下的集群存在单点故障，不适用于正式的产品环境。而称多于 1 个 Server 节点的（通常为单数，3 个或 5 个）Consul 集群为“集群模式”，只有这种配备才真正适用于生产。集群的 Server 角色节点个数需要在创建时决定，并通过 `-bootstrap-expect` 参数在启动时给出。下面的命令将创建单节点的 Consul 服务。

```
$ consul agent -server -bootstrap-expect=1 \
  -data-dir=/opt/data.d -config-dir=/etc/consul.d \
  -bind=0.0.0.0 -http-port=8500 -dns-port=8600 \
  -node=server-agent-1
```

Consul 启动后会占用系统的 8300、8301、8302、8400、8500、8600 一共 6 个端口，但其中只有提供 HTTP 服务的 8500 端口和提供 DNS 服务的 8600 端口是可以通过启动参数修改。若要修改其他的服务端口，则需要使用“cluster.json”配置文件，将它放到启动时

^① <https://www.consul.io/downloads.html>

-config-dir 参数指定的目录里，如下所示。

```
$ cat <<EOF | sudo tee /etc/consul.d/cluster.json
{
  "datacenter": "dc1",
  "data_dir": "/opt/data.d",
  "log_level": "INFO",
  "node_name": "server-agent-1",
  "client_addr": "0.0.0.0",
  "server": true,
  "ports": {
    "dns": 8600,
    "http": 8500,
    "rpc": 8400,
    "server": 8300,
    "serf_lan": 8301,
    "serf_wan": 8302
  }
}
EOF
```

然后使用更简短的命令来启动服务，如下所示。

```
$ consul agent -config-dir=/etc/consul.d -bootstrap-expect=1
```

与 Etcd 一样，从控制台启动 Consul 时，它会保持在前台运行，建议通过 Systemd 将它作为系统服务，或使用 Docker 部署。

如果是创建集群模式的 Consul 服务，只需要将 -bootstrap-expect 参数的值改为实际的节点个数，然后在多个节点上分别用不同的 -node 参数值运行这个命令。值得注意的是，在创建 Consul 节点时，并不需要像 Etcd 那样告诉它其他 Server 节点的地址，因为稍后会用 consul join 命令将它们组织在一起，如下所示。

```
# 节点 1
$ consul agent -server -bootstrap-expect=3 \
  -data-dir=/opt/data.d -config-dir=/etc/consul.d \
  -bind=0.0.0.0 -http-port=8500 -dns-port=8600 \
  -node=server-agent-1

# 节点 2
$ consul agent -server -bootstrap-expect=3 \
  -data-dir=/opt/data.d -config-dir=/etc/consul.d \
  -bind=0.0.0.0 -http-port=8500 -dns-port=8600 \
  -node=server-agent-2
```

```
# 节点3
$ consul agent -server -bootstrap-expect=3 \
  -data-dir=/opt/data.d -config-dir=/etc/consul.d \
  -bind=0.0.0.0 -http-port=8500 -dns-port=8600 \
  -node=server-agent-3
```

然后启动任意多个 Client 模式的 Consul 节点，启动方式与 Server 模式节点基本相同，仅仅是需要 `-server` 和 `bootstrap-expect` 参数，如下所示。

```
$ consul agent -data-dir=/opt/data.d -config-dir=/etc/consul.d \
  -bind=0.0.0.0 -http-port=8500 -dns-port=8600 \
  -node=client-agent-1
```

最后在任意一个 Server 节点上，使用 `consul join` 将其他的 Server 和 Client 节点添加进来，如下所示。

```
$ consul join <其他节点的 IP 地址>
```

刚刚加入时，只有执行 `consul join` 的节点知道整个集群的结构，其他节点都只知道拉它入伙的那个节点的存在。接下来 Consul 通过 Gossip 协议（中文有“小道消息”的意思）让所有节点自动从已经识别的节点学习集群的信息，最终使得所有节点都获取到整个集群的状态。这正是 Gossip 协议名字的由来。使用 `consul members` 命令可以查看当前节点识别到的 Consul 集群成员状态，如下所示。

```
$ consul members
Node           Address Status Type    Build  Protocol DC
client-agent-1 ...:8301 alive  client 0.8.4  2       dc1
...
server-agent-1 ...:8301 alive  server 0.8.4  2       dc1
...
```

Consul 的键值操作与 Etcd 基本相似，相关的命令有 `consul kv`、`consul watch` 等，如下所示。

```
# 添加或修改键值
$ consul kv put demo "hello consul"
Success! Data written to: demo

# 获取键值内容
$ consul kv get demo
hello consul

# 获取键值的详细属性
$ consul kv get -detailed demo
```

```
CreateIndex    7
Flags          0
Key            demo
LockIndex      0
ModifyIndex    7
Session        -
Value          hello consul
```

```
# 删除指定键值记录
$ consul kv delete demo
Success! Deleted key: demo
```

Consul 中的键可以像文件系统那样形成层级，如下所示。

```
$ consul kv put dir/zookeeper "hello zookeeper"
$ consul kv put dir/etcd "hello etcd"
$ consul kv put dir/consul "hello consul"

$ consul kv get -recurse dir
dir/consul:hello consul
dir/etcd:hello etcd
dir/zookeeper:hello zookeeper
```

监控一个键的变化，如下所示，当发生变化时执行指定的脚本。不过在执行脚本时 Consul 并不会把变化的内容传给执行的脚本，如果想要获得变化的情况，需要在脚本中重新读取一次 Consul，若多个监控共用同一个脚本，则无法判断出发生变化的是哪一个键，坦白来说这个命令其实不太实用。

```
$ consul watch -type=key -key=demo /path/to/change-handler.sh
```

Consul 中其他对键值的相关操作与 Etcd 大同小异，这里不再赘述。下面来看一下 Consul 的一些有特色的功能。

首先是服务发现。在 Consul 中注册服务有两种方式，一种是将服务的详细信息直接通过 HTTP 接口发送给 Consul，让 Consul 记录下来。这种方式与把服务信息记入普通键值存储相比，多了一层格式上的约束，即要求服务注册时必须按照指定的格式提供信息，这层约束确保了 Consul 的域名服务、根据节点搜索等功能的正常使用，同时也使得进行服务发现时对取到的服务信息有一个提前预期（比如每个服务一定会有名字、端口、节点 IP、标签等标准数据），避免因不同服务采用不同记录格式而导致混乱。另一种方式是 Consul 比较推荐的，即在所有需要运行服务节点上运行一个 Consul Agent，然后每个服务启动时在 Consul 的配置目录下创建一个自己的 Json 格式服务描述文件，这个文件只需提供服务的名

字、标签、端口等很少的信息，由 Consul 自动补全与运行节点相关的内容。

例如在集群的某个节点上运行了一个 MySQL 数据库服务，若要将它注册到 Consul，只需在该节点的 Consul 配置目录（例如“/etc/consul.d”）创建一个 Json 描述文件，如下所示。

```
$ cat <<EOF >/etc/consul.d/mysql-dev.json
{
  "service": {
    "name": "mysql",
    "tags": ["dev"],
    "port": 3306
  }
}
EOF
```

每次添加或更新服务描述文件后，都要给节点的 Consul Agent 发一个 SIGHUP 信号，让它重新加载配置目录，如下所示。

```
$ kill -SIGHUP <ConsulAgent 进程的 PID>
```

这样就完成了服务的注册。在任意 Consul 集群的任意一个节点上都可以通过服务查询 API 找到它，如下所示。

```
$ curl -s http://localhost:8500/v1/catalog/service/mysql | jq .
[
  {
    "ID": "266b58ec-36a2-ebad-db31-45e24f68ad36",
    "Node": "server-agent-1",
    "Address": "192.168.33.11",
    "Datacenter": "dc1",
    "TaggedAddresses": {
      "lan": "192.168.33.11",
      "wan": "192.168.33.11"
    },
    "NodeMeta": {},
    "ServiceID": "mysql",
    "ServiceName": "mysql",
    "ServiceTags": [ "dev" ],
    "ServiceAddress": "",
    "ServicePort": 3306,
    "ServiceEnableTagOverride": false,
    "CreateIndex": 438,
    "ModifyIndex": 438
  }
]
```


通过 API 也可以查看所有已注册服务的清单，如下所示。

```
$ curl -s http://localhost:8500/v1/catalog/services | jq .
{
  "consul": [ ],
  "mysql": [ "dev" ]
}
```

Consul 的另一个特色功能是 DNS 域名解析服务，集群中所有的节点和服务都会有一个以自身名称为前缀的唯一域名。Consul 的 DNS 服务默认运行在每个节点的 8600 端口。可以使用 `nslookup` 或 `dig` 命令来查询这些 DNS 记录，如下所示。

```
# 查询指定节点的 DNS 记录
$ dig @127.0.0.1 -p 8600 server-agent-1.node.dc1.consul
...
;; ANSWER SECTION:
server-agent-1.node.dc1.consul. 0 IN      A      192.168.33.11
...

# 查询指定服务的 DNS 记录
$ dig @127.0.0.1 -p 8600 dev.mysql.service.consul SRV
...
;; ANSWER SECTION:
dev.mysql.service.consul. 0 IN  SRV 1 1 3306 server-agent-1.node.dc1.consul.
;; ADDITIONAL SECTION:
server-agent-1.node.dc1.consul. 0 IN      A      192.168.33.11
...
```

Consul 对服务的健康检查也是与服务发现紧密相关的。在注册服务时，有一个可选的额外属性 `check`，例如每隔 30s 使用指定的命令检查一次服务是否运行（注意修改服务描述后需要用 `SIGHUP` 信号通知 Consul 重载配置），如下所示。

```
$ cat <<EOF >/etc/consul.d/mysql-dev.json
{
  "service": {
    "name": "mysql",
    "tags": ["dev"],
    "port": 3306,
    "check": {
      "script": "nc -zw1 127.0.0.1 3306",
      "interval": "30s"
    }
  }
}
EOF
```


根据检查命令的返回值，Consul 将服务状态分为 `passing`（返回值为 0）、`warning`（返回值为 1）、`critical`（其他返回值）三种状态。健康检查的结果会影响服务的 DNS 查询域名，如果服务状态不是 `passing`，在查询 DNS 时将不会返回该记录的信息。这样若有多个节点同时运行了某项服务，健康检查将能够确保用户始终只得到运行正常的节点的信息。

通过 Consul 的 API 能够查看指定服务的健康检查结果，也能查看处于指定状态的所有服务列表，如下所示。

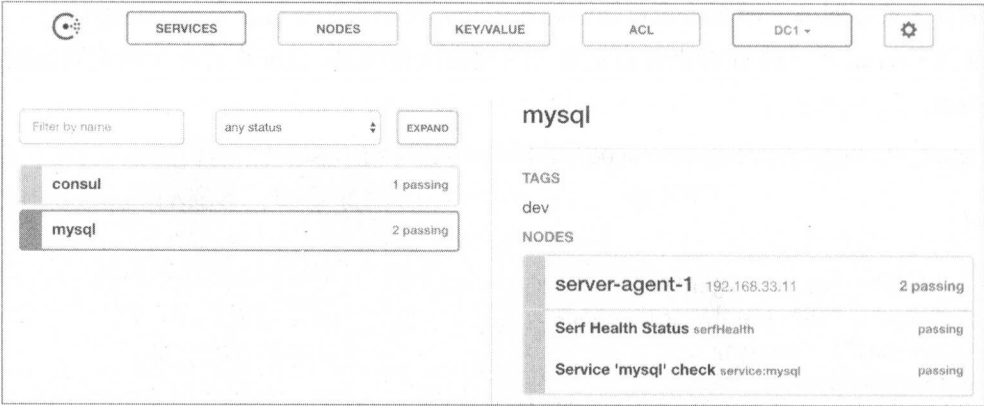
```
$ curl -s http://localhost:8500/v1/health/checks/mysql
[
  {
    ...
    "Node": "client-agent-1"
    "Name": "Service 'mysql' check",
    "Status": "warning",
    "ServiceName": "mysql",
    "ServiceTags": [ "dev" ],
    ...
  }
]

$ curl -s http://localhost:8500/v1/health/state/warning
[
  {
    "Node": "client-agent-1",
    "CheckID": "service:mysql",
    "Name": "Service mysql check",
    "Status": "warning",
    "ServiceName": "mysql",
    "ServiceTags": [ "dev" ],
    ...
  },
  { ... }
]
```

目前 Consul 的命令行工具还算不上完善，许多操作依然需要通过 API 才能完成。不过对于日常的查询和监控等工作，Consul 提供了一个简单易用的图形界面，支持包括服务、节点、键值、数据中心和访问规则的管理，足以弥补它在命令行功能上的不足。出于安全等因素的考虑，Consul 的图形界面功能默认是关闭的，如果需要开启，需要在启动 Agent 时添加 `-ui` 参数，如下所示。

```
$ consul agent -dev -ui
```

注意，图形界面只会在使用了“-ui”参数启动的节点上可用，而不是整个集群。使用浏览器访问这个节点地址 8500 端口的/ui 路径，例如“http://127.0.0.1:8500/ui”，就会打开如图 7-17 所示的图形界面。



7-17 Consul 的图形界面

相比 ZooKeeper 或 Etcd 这样的通用键值存储记录服务信息实现的服务注册中心，Consul 专用的服务注册发现 API 无疑提供了更全面和规范性的操作体验，但它的代价则是更高的复杂性和对 Consul 服务模型的依赖。此外，随着微服务架构的流行，有些适用于微服务开发的框架同样包含了特定的服务注册中心，例如 Spring Cloud 中的 Eureka 服务，在具体的项目中可根据实际情况进行选择。

7.4 镜像仓库

7.4.1 容器镜像仓库概述

发布包的版本仓库的概念并非源于容器或 Docker 之手，在软件开发领域，为了解决带发布包的构建和部署的时间、位置不同的问题，很早就有了包仓库的概念。例如 Java 的 maven 仓库、Python 的 pip 仓库、Ruby 的 gem 仓库，甚至 Linux 中的 apt 和 yum 也通过统一仓库来分发和管理各种软件和依赖。

第 1 章提到过，容器所使用的技术很早就根植于 Linux 内核中，Docker 之所以流行，并不是因为它发明了新的容器技术，而是因为它创造了一种独特的基于容器的交付方式。

Docker 的创造者将它归纳为三个步骤：Build、Ship、Run（构建、运输、运行）。也就是说，首先通过 Dockerfile 将构建过程标准化、代码化，然后将产生的镜像推送到指定的镜像仓库，最后在有要运行服务的地方从仓库获取镜像并启动。在 Docker 之后的容器工具（例如 Rkt、LXD）基本也都采纳了这种通过镜像仓库管理镜像发布的思路。

尽管不同容器工具的镜像结构不尽相同，OCI（开放容器组织）已经为容器的镜像制定了统一的标准，这个标准参照了当前 Docker 镜像的结构和实现。意味着 Docker 的容器仓库未来将能为各种其他容器技术提供统一的镜像版本管理服务。

对于许多开发者来说，最常打交道的镜像仓库莫过于 Docker Hub 了。作为 Docker 镜像的官方仓库，Docker Hub 存储了海量的免费镜像资源，普通用户只需注册一个账号，就能享受免费且不限量的公有镜像上传服务。但这些公有镜像可以被任何人访问到，而在 Docker Hub 上创建私有镜像是需要付费的。国内的一些容器平台提供商，例如阿里云、灵雀云等也提供了国内访问更快、更适于国内用户使用的公有镜像仓库服务。

在真实的应用场景里，除费用、网速和安全性等的顾虑，有许多企业内部的网络环境都没有与因特网连接，无法享用 Docker Hub 或者其他公用镜像仓库带来的便利。因此自建私有镜像仓库（特别是内网私有镜像仓库）的需求是十分迫切的。

目前，Docker 并没有开源它的镜像仓库服务实现，开源社区曾经出现过一些第三方实现的 Docker 镜像仓库服务项目，也都没有最终持续下去。没开源归没开源，Docker 还是提供了免费的仓库镜像仓库的镜像“Registry”，发布在 Docker Hub 上，使得用户自行搭建起一个简单可用的私有仓库也并非什么难事。不过官方的免费镜像没有提供管理界面和权限管理等高级功能，而是在付费的企业级版本（Docker Enterprise Edition）中另外提供了一款功能完善的“Docker Trusted Registry”服务。从某种程度上说，Docker 这一点的小私心对准备采用容器发布的中小型企业来说挺要命，免费版仓库满足不了基本的安全需求，而完善的企业版仓库价格昂贵。

所幸的是，除了官方的解决方案，免费但同样非开源的通用软件包仓库项目 Nexus 3 提供了功能比较全面的 Docker 镜像仓库功能，可以作为 Docker Trusted Registry 以外的一种替代。此外，VMware 基于官方的免费“Registry”镜像设计了一款功能比较全面的企业级镜像仓库 Harbor，并将额外增加的辅助服务全部开源。

接下来的内容将介绍一些使用官方镜像仓库 Registry 和企业级镜像仓库 Harbor 的技巧。

7.4.2 Registry

官方的 Registry 仓库使用很简单，启动容器并将其 5000 端口映射到主机即可，如下所示。

```
$ docker run -d -p 5000:5000 registry:2
```

在实际使用时一般还会将用于存储镜像数据的目录挂载到主机，以获得更好的读写效率，并避免在重建 Registry 容器时（例如版本升级）丢失所有已保存的镜像。为容器加上有意义的名称和自动重启属性也是常见的好习惯，如下所示。

```
$ docker run -d --name registry --restart always \
-v /path/to/registry_data_folder:/var/lib/registry \
-p 5000:5000 \
registry:2
```

默认的配置下，Registry 仓库提供的是 HTTP 协议的服务，而 Docker 只信任 HTTPS 协议的非 localhost 地址仓库。因此在使用这种简易仓库时，需要为 Docker 后台服务添加 `--insecure-registry` 启动参数，以表示对这个仓库的读写是使用非加密协议进行的，如下所示。

```
/usr/bin/dockerd --insecure-registry <Registry 服务节点的 IP 地址>:5000
```

为了能让仓库使用安全的协议，过去通常是在 Registry 服务的前级加上一道 Apache 或者 Nginx 反向代理，让这个反向代理完成 HTTP 和 HTTPS 协议的转换。较新版本的 Registry 仓库已经内置了 HTTPS 协议的实现，要使用此功能，只需提供适当的根证书和密钥对，通过环境变量在启动时注入 Registry 服务的运行上下文即可。Registry 会检查到环境变量的存在，然后自动修改服务配置，使得用户能够比较容易地获得安全地镜像仓库服务，如下所示。

```
$ mkdir certs

$ openssl req \
  -newkey rsa:4096 -nodes -sha256 -keyout certs/ca.key \
  -x509 -days 365 -out certs/ca.crt

$ docker run -d --name registry --restart=always \
-v /path/to/registry_data_folder:/var/lib/registry \
-v `pwd` /certs:/certs \
-e REGISTRY_HTTP_ADDR=0.0.0.0:443 \
-e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/ca.crt \
-e REGISTRY_HTTP_TLS_KEY=/certs/ca.key \
-p 443:443 \
registry:2
```

在生成证书时，Openssl 命令会提一系列问题。当问到“Common Name”时，应该输入一个有效的域名，并且这个域名已经指向用于运行 Registry 服务的节点的 IP 地址，因为

证书是要与域名绑定的。

由于使用了自签名的密钥文件，仍然需要手动地在各个使用仓库的节点上将该签名证书添加为受信任的根证书（相比之下，这种修改至少比在每个节点上修改 Docker 后台服务的启动参数要简单一点）。将生成的 ca.crt 根证书拷贝为“/etc/docker/certs.d/<完整的域名和端口>/ca.crt”文件，其中“完整的域名和端口”可以是“registry-demo.com:5000”，若使用的是 HTTPS 的默认 443 端口，则可以省略端口号，如下所示。

```
$ sudo mkdir /etc/docker/certs.d/<完整的域名和端口>
$ sudo cp certs/ca.crt /etc/docker/certs.d/<完整的域名和端口>/ca.crt
```

所有使用这个仓库的主机都需要通过这种方式添加根证书的信任。

当然，你也可以选择权威证书机构签发的受信任密钥或是使用例如“Let's Encrypt”^①提供的（90 天有效性的）免费受信任密钥文件。由于这种密钥签发的根证书是 Docker 内置信任的，可以免去在所有主机上添加根证书信任的操作。

除了采用安全的协议，若将 Registry 仓库服务的端口发布到公网使用，还会存在访问安全的问题。Registry 内置了基于 HTTP 请求头（silly 模式）、外部效验（token 模式）或密码文件（htpasswd 模式）的简单鉴权机制。以密码文件为例，较新版本的 Registry 镜像中内置有 htpasswd 命令，可以用于生成用户名和密码序列，如下所示。

```
$ mkdir auth

$ docker run \
  --entrypoint htpasswd \
  registry:2 -Bbn testuser testpassword > auth/htpasswd

$ docker run -d \
  -v /path/to/registry_data_folder:/var/lib/registry \
  --restart=always \
  --name registry \
  -v `pwd`/auth:/auth \
  -e "REGISTRY_AUTH=htpasswd" \
  -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" \
  -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
  -v `pwd`/certs:/certs \
  -e REGISTRY_HTTP_ADDR=0.0.0.0:443 \
  -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/ca.crt \
  -e REGISTRY_HTTP_TLS_KEY=/certs/ca.key \
  -p 443:443 \
  registry:2
```

① <https://letsencrypt.org>

客户端需要先用 `docker login` 登录，然后方可正常使用镜像仓库服务，如下所示。

```
$ docker login -u=testuser -p=testpassword <完整的域名和端口>
```

这种鉴权方法配置简单（和需要外部鉴权系统的 token 模式相比），安全性较高（和只要请求有 Authorization 消息头就放行的 silly 模式相比）。生成密码文件中的每一行都是一个用户的登录信息，对密码文件内容的修改会实时生效，足以应对普通镜像仓库安全需求，但如果用户数量较多，管理起来还是比较麻烦。对于更复杂的用户管理和鉴权需求，例如与企业账号拉通等，就需要考虑换用支持活动目录和 LDAP 等鉴权集成的企业级镜像仓库产品，例如 Harbor 或 Docker Trusted Registry。

Registry 仓库的镜像存储和查询等功能实质上都是通过一套标准 HTTP API 提供的（目前的版本是 v2），然而这些 API 的粒度比较细，主要的操作资源是 blobs 和 manifests。每个 blobs 可以看作镜像中的一层，而 manifests 是代表镜像的元数据，其中包含了通过 blobs 组成镜像的信息，要是不借助 Docker 客户端的镜像拼装逻辑，手动调用仓库 API 来上传和获取镜像是十分有难度的事情。

值得一说的是，镜像仓库在使用一段时间后难免会遇到空间不足的问题，此时就需要删除一些不再使用的过时镜像，以释放空间给新的镜像使用。在 Registry 仓库的 API 中，与删除镜像比较相关的只有删除 manifests 的操作。然而这个操作实际上只会删除仓库中的镜像元数据，并不包括镜像的实体，也就是 blobs 数据（因为同一个 blobs 可能还会被其他镜像引用），这个操作不会减少占用的存储空间。为了真正达到释放空间的目的，可以使用 `garbage-collect` 参数运行在 Registry 容器里的 `registry` 命令，如下所示。

```
$ docker exec registry \
  bin/registry garbage-collect /etc/docker/registry/config.yml
```

这个命令会遍历当前仓库中的所有 blobs，将没有任何来自镜像或其他 blobs 引用的孤立 blobs 块物理删掉。这个操作需要用户指定 Registry 仓库的配置文件，即以上命令中的“`config.yml`”文件。

这个“`config.yml`”文件其实是 Registry 仓库的核心配置文件。此外，在创建 Registry 容器时传入的各种环境变量，例如“`REGISTRY_AUTH`”和“`REGISTRY_HTTP_TLS_KEY`”等，也能改变 Registry 的配置，并且具有比“`config.yml`”文件更高的优先级，如下所示。

```
$ docker exec registry cat /etc/docker/registry/config.yml
version: 0.1
log:
  fields:
    service: registry
```

```

storage:
  cache:
    blobdescriptor: inmemory
  filesystem:
    rootdirectory: /var/lib/registry
http:
  addr: :5000
  headers:
    X-Content-Type-Options: [nosniff]
health:
  storagedriver:
    enabled: true
    interval: 10s
    threshold: 3

```

这个配置文件的所有可用属性在 Docker 的官方文档^①里有详细描述。表 7-3 列举了其中的所有顶级配置项和相应的功能。

表 7-3 Registry 服务的顶级配置项

配置项	描述
version	必须存在的属性，配置文件格式的版本，目前最新版本为“0.1”
log	与仓库服务本身日志的打印级别、格式和内容相关配置
hooks	在仓库服务本身出现故障时自动发送告警邮件的功能配置
storage	必须存在的属性，配置镜像内容的存储位置和参数，此外还包括缓存、是否允许删除等存储相关参数
auth	用户鉴权配置，包括 silly、token、htpasswd 三种模式
middleware	提供一些第三方扩展中间件的配置选项，例如基于 AWS 的 CDN 服务 CloudFront 来缓存镜像的查询和读取功能
reporting	在仓库服务本身出现错误时，生成错误信息报告的功能，目前支持将报告对接到 Bugsnag 或 New Relic 平台
http	必须存在的属性，服务监听的地址、端口、域名、密钥证书等与 API 访问相关的配置
notifications	将仓库服务的事件信息推送到第三方平台的功能配置
redis	用来提高读取镜像数据吞吐量的 Redis 缓存配置
health	与仓库服务自身健康检查相关的配置
proxy	将仓库服务作为仓库代理时需要的配置
compatibility	用于处理旧版本配置格式的结构映射相关的预留配置
validation	访问规则配置，可以禁用对特定名称特征的镜像的访问

镜像的存储位置默认为本地目录，还支持 AWS S3、Openstack Swift、Google Cloud

① <https://docs.docker.com/registry/configuration/>

Storage、Azure Blob Storage 和 Aliyun OSS 存储等在线存储服务。利用云端无限存储空间的能力，对于大型镜像仓库而言能够减少很多扩容和维护存储资源方面的工作。

除了作为真实存储镜像的私有仓库，Registry 还可以作为仓库代理，用于内网访问 DockerHub 或其他外部仓库的加速器。工作在代理模式的仓库只能用于读取镜像，不能写入，所有获取镜像内容的请求都会被透明地代理到另一个远程镜像仓库，并缓存在本地，若下一次有用户请求相同的镜像时，则可以直接从本地缓存的镜像返回，从而达到加速内网访问外部仓库速度的目的。

首先创建一个包含 proxy 属性的 Registry 仓库配置文件，如下所示。

```
$ cat <<EOF >config.yml
version: 0.1
log:
  fields:
    service: registry
storage:
  cache:
    blobdescriptor: inmemory
  filesystem:
    rootdirectory: /var/lib/registry
http:
  addr: 0.0.0.0:443
  net: tcp
  tls:
    certificate: /certs/ca.crt
    key: /certs/ca.key
  headers:
    X-Content-Type-Options: [nosniff]
health:
  storagedriver:
    enabled: true
    interval: 10s
    threshold: 3
proxy:
  remoteurl: https://registry-1.docker.io
EOF
```

然后创建 Registry 服务并用这个文件覆盖容器中原有的配置文件，由于监听端口和 SSL 密钥证书也已经通过配置文件给出，因此不再需要指定在环境变量中，如下所示。

```
$ docker run -d --restart=always --name registry \
-v `pwd`/config.yml:/etc/docker/registry/config.yml \
-v /path/to/registry_data_folder:/var/lib/registry \
```

```
-v `pwd`/certs:/certs \
-p 443:443 \
registry:2
```

配置中使用的 `registry-1.docker.io` 地址实际上是 DockerHub 仓库的 API 地址，因此只需在官方仓库中存在的镜像名称前加上私有仓库的“完整的域名和端口”（若端口为标准的 443 则可省略）就可以通过私有仓库代理获得相应的镜像。例如私有仓库服务器的域名为“`registry-demo.com`”，下面这个命令将获得官方的 Ubuntu 容器镜像。

```
$ docker pull registry-demo.com/library/ubuntu
```

这种做法虽然能够加速内网访问外部镜像的体验，但带来了少许的副作用，其一是每次拉取镜像时要在镜像名称前面加上一段私有仓库地址，其二是获得的镜像名称也会带有私有仓库地址，略显不美观。有没有一劳永逸的办法能把代理的工作隐藏起来呢？

答案是为 Docker 的后台服务使用“仓库代理”（`registry-mirrors`）参数，具体的设置方法有两种，一种是直接为 Docker 后台服务添加 `--registry-mirror` 启动参数，如下所示。

```
/usr/bin/dockerd --registry-mirror=https://<完整的域名和端口>
```

或是创建“`/etc/docker/daemon.json`”配置文件，内容如下。

```
{
  "registry-mirrors": ["https://<完整的域名和端口>"]
}
```

不论是哪种方式，修改完成后都需要重启一次 Docker 后台服务（如果是 Systemd 托管的服务，还需要先执行 `sudo systemctl daemon-reload` 重新加载一次修改过的服务配置）。在所有需要使用镜像代理的节点上，都需要做这样的配置。

7.4.3 Harbor

Harbor^①是由 VMware 公司的工程师开发并开源的一套企业级容器仓库产品，它复用了官方的 Registry 仓库服务，并在此基础上扩展出用户分组、安全鉴权、Web 界面、镜像同步、日志管理等丰富功能。

从结构上看，Harbor 项目由多个独立运行的子服务组成，如下所示。

- `harbor-log`: 统一日志管理服务，采集 Harbor 所有服务的日志。

① <http://vmware.github.io/harbor/>

- registry: 镜像仓库服务，实际上就是 Docker 官方的 Registry 服务。
- harbor-db: 存储项目、用户和权限数据的 MySQL 数据库。
- harbor-adminserver: 提供 Harbor 的系统管理接口，用于修改系统配置和获取系统信息。
- harbor-ui: Harbor 的 Web 管理页面，便于用户操作。
- harbor-jobservice: 管理任务的服务，主要是为了镜像仓库之间同步。
- nginx: 反向代理，负责流量转发和安全验证。

这些服务之间的关系大致如图 7-18 所示。

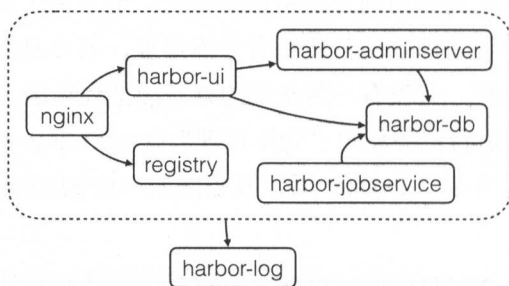


图 7-18 Harbor 服务间关系

部署 Harbor 镜像仓库并不复杂，首先从它的 GitHub 发布页面^①下载最新版本的安装包，如下所示。

```
$ wget https://github.com/vmware/harbor/releases/download/<version>/
harbor-offline-installer-<version>.tgz
```

解压并进入新生成的“harbor”目录，如下所示。

```
$ tar xzf harbor-offline-installer-<version>.tgz
$ cd harbor
```

准备 SSL 证书和密钥文件，也可以重新生成自签名的证书并签发密钥。例如假设私有仓库服务器的域名为“registry-demo.com”，虽然可以直接使用根密钥，但还是建议为仓库域名创建单独的子密钥文件，如下所示。

```
$ mkdir certs
$ openssl req \
    -newkey rsa:4096 -nodes -sha256 -keyout certs/ca.key \
    -x509 -days 365 -out certs/ca.crt
```

^① <https://github.com/vmware/harbor/releases>


```
$ openssl req \
  -newkey rsa:4096 -nodes -sha256 \
  -keyout certs/registry-demo.com.key \
  -out certs/registry-demo.com.csr
$ openssl x509 -req \
  -in certs/registry-demo.com.csr \
  -CA certs/ca.crt -CAkey certs/ca.key \
  -CAcreateserial -days 365 -out certs/registry-demo.com.crt
```

同样地，自签名的密钥需要明确地添加根证书到每个节点的密钥信任列表中，如下所示。

```
$ sudo mkdir /etc/docker/certs.d/registry-demo.com
$ sudo cp certs/ca.crt /etc/docker/certs.d/registry-demo.com/ca.crt
```

然后依据实际情况修改“harbor”目录中的“harbor.cfg”文件中的以下几行配置。

```
hostname = registry-demo.com      #仓库节点域名
ui_url_protocol = https           #建议使用 HTTPS
db_password = your-secret-passwd  #MySQL 数据库密码
ssl_cert = /path/to/cert/registry-demo.com.crt
ssl_cert_key = /path/to/cert/registry-demo.com.key
```

Harbor 默认采用 Docker Compose 部署（也可以用 Kubernetes 部署），因此。在“<https://github.com/docker/compose/releases>”页面查找到最新的 docker-compose 版本，然后将它下载到服务器的“/usr/local/bin/”目录，如下所示。

```
$ sudo curl -L -o /usr/local/bin/docker-compose https://github.com/docker/
compose/releases/download/<版本号>/docker-compose-`uname -s`-`uname -m`
$ sudo chmod +x /usr/local/bin/docker-compose
```

最后执行 harbor 目录下的“install.sh”脚本完成 Harbor 服务的部署，如下所示。

```
$ sudo ./install.sh
```

此时使用 HTTPS 地址访问这个节点的 443 端口，就会看到 Harbor 的登录界面。初次访问时可以默认的管理员账号 admin 登录，密码为 Harbor12345。如图 7-19 所示。

从右上角可以切换界面语言为中文。

Harbor 的镜像组织结构与 DockerHub 有几分相似，仓库中默认创建了一个名称是“library”的公有镜像组（在 Harbor 称为“项目”），并且仓库中的每个镜像都会属于某个项目。项目可以是公有或私有的，公有项目中的镜像不需要登录就可以下载，而私有项目必须先 docker login 后才能下载，不论是公有还是私有的项目，上传镜像之前都需要先使用 docker login 登录。



图 7-19 Harbor 的主页面

假设镜像仓库的服务器域名是“registry-demo.com”，用户若要将自己的镜像上传到仓库，需要先在 Harbor 上创建或加入一个项目，例如称为“demo”，如下所示。然后将镜像重命名为服务 Docker 标准的“<仓库地址>/<镜像组>/<镜像>”规则，登录仓库，就可以执行 docker push 了。

```
$ docker login -u <用户名> registry-demo.com
$ docker tag alpine registry-demo.com/demo/alpine
$ docker push registry-demo.com/demo/alpine
```

基于用户的授权和安全机制是 Harbor 提供的一种重要功能。在 Harbor 中有两类用户，一类为管理员，另一类为普通用户。管理员可以对系统配置和用户进行管理，不论是普通用户还是管理员都可以作为镜像组（项目）的成员，进行上传和下载镜像。

每个用户除了在系统中的权限类型，在作为每个镜像组的成员时也可以划分为三种角色，分别是管理员、开发者、访客。访客可以查看和读取镜像，但不能修改或向项目中提交新的镜像，开发者能够读取和提交镜像，管理员则可以修改其他各用户在当前项目中所属的角色，甚至将用户移出项目。这种角色划分对应于项目团队中的项目经理、产品经理、开发、测试、运维等职能所需的不同权限。例如，开发人员需要拥有对镜像的读写（PULL/PUSH）权限以更新镜像版本，测试和运维人员需要有读取（PULL）镜像的权限，而项目经理需要对上述的角色进行管理。

用户管理的菜单在系统管理员用户面板左侧的“系统管理→用户管理”中，而项目的成员则通过在各个项目页面的“成员”标签页中进行管理。“Harbor”中的用户来源除了直接在本系统中创建和管理，也可和 LDAP 对接，只需将“harbor.cfg”文件的 auth_mode

配置值修改为 `ldap_auth`，并按照实际情况填写配置文件中有关 LDAP 服务端的配置，例如 `ldap_url`、`ldap_basedn` 等。修改过登录方式后，需要重建 Harbor 的服务容器，先执行 `docker-compose stop`，然后重新执行“`install.sh`”脚本即可。可以放心，仓库中的镜像和用户等数据不会由于重建容器而丢失。

对于比较大型的组织，单个的镜像仓库服务往往难以承受所有用户的镜像获取需求。此时推荐的做法是为仓库配置读写分离，也就是说利用 Harbor 的仓库同步功能，让所有的镜像写入主仓库，但在读取镜像时，从多个同步仓库里获得镜像数据。这种结构如图 7-20 所示。

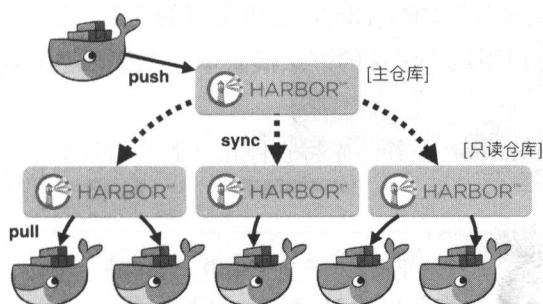


图 7-20 镜像仓库的读写分离

Harbor 的复制是基于项目的，登录主仓库后进入需要复制的项目页，单击“Replication（复制）”标签，然后单击“+Replication Rule（复制规则）”按钮，如图 7-21 所示。



图 7-21 项目的复制任务管理页面

在提示框中输入规则名称、复制的目标仓库地址和登录用户等信息，勾选“启用（Enable）”选项框，然后单击“确认”就创建了一个镜像复制规则。在界面上单击任意一

条复制规则，会在下方的任务列表中看到当前规则执行的任务，每个任务代表了一次镜像同步的操作。在主仓库界面左侧菜单的“系统管理→复制管理”中，能看到所有已添加的复制目标和复制规则，在这里可以方便地管理复制目标，以及禁用、启用、编辑复制规则。

登录到作为复制目标的 Harbor 仓库，会看到仓库里创建了与主仓库同名的项目，在项目中包含与主仓库完全一致的镜像列表。为仓库创建多个复制目标，加上额外的负载均衡器，不仅能统一读取镜像的仓库入口地址，还能实现镜像读取时的高可用。

使用过企业版 Docker Trusted Registry 服务的读者可能会知道它与免费的 DockerHub 相比，额外提供了镜像的漏洞扫描功能，用于发现用户镜像中潜在的问题。Harbor 从其 1.2 版本开始集成了镜像漏洞扫描服务 Clair^①，提供类似的功能。默认方式启动的 Harbor 是不含有 Clair 组件的，不过启用这个方法很简单，只需在运行 `install.sh` 脚本时加上 `--with-clair` 参数。

若之前已经运行了 Harbor 服务，先将它停止（若是首次启动则忽略这个操作），如下所示。

```
$ docker-compose down
```

加上 `--with-clair` 参数重新启动 Harbor，如下所示。

```
$ sudo ./install.sh --with-clair
```

再次进入 Harbor 后，在“系统管理→配置管理”菜单会看到一个新增加的“Vulnerability（漏洞）”标签，如图 7-22 所示。

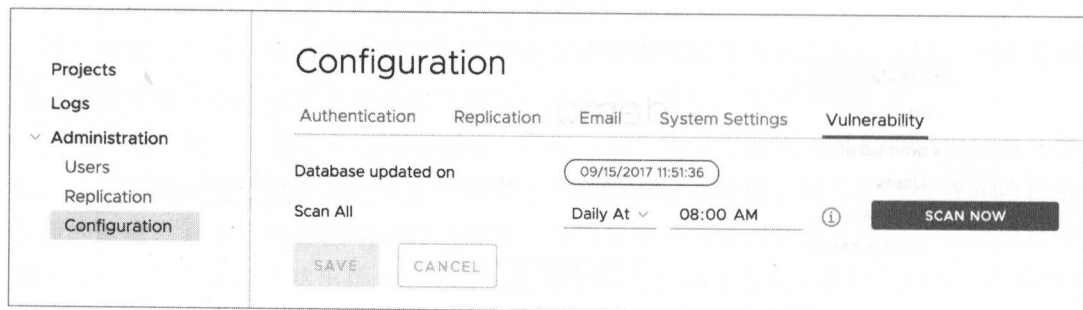


图 7-22 镜像漏洞扫描配置

这里可以配置执行自动镜像扫描的频率以及立即执行扫描的按钮。单击“开始扫描”然后回到任意项目的镜像列表页面，会看到每个镜像都包含了一个安全扫描结果，如图 7-23

^① <https://github.com/coreos/clair>

所示。单击镜像的任意一个版本都能查看所有扫描出漏洞的详细信息。

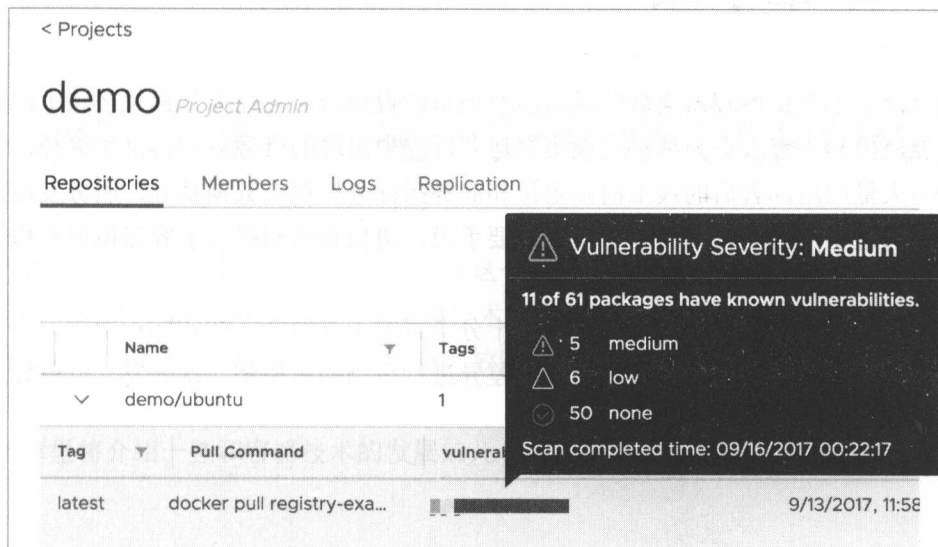


图 7-23 镜像的安全扫描结果

由于 Clair 的扫描规则比较细致，很少有镜像能在扫描后毫无问题，不过其实许多镜像中都包含了一些内核相关文件，主要用于应对镜像中的服务启动时需要检测依赖的情况，在实际运行的时候并不会使用它们（而是与系统共享内核），这其中扫描出的很多问题都是不需要修复的。实际使用时可以对比用户自建的服务与所使用的基础镜像之间是否引入了新的漏洞，若有则需要引起重视，因为它们很可能是镜像中的服务真实存在的安全问题。

值得一说的是，使用 `--with-clair` 参数启动的 Harbor 服务实际上使用了两个 Docker Compose 服务编排文件，即 Harbor 安装目录下的“`docker-compose.yml`”和“`docker-compose.clair.yml`”。因此若要通过 `docker-compose` 重启使用了 Clair 功能的 Harbor 服务（比如修改了“`harbor.cfg`”配置文件内容后），应该依次显示指定的两个编排文件，如下所示。

```
$ docker-compose -f docker-compose.yml -f docker-compose.clair.yml restart
```

除了 Clair 以外，Harbor 还支持 Docker 镜像签名组件 Notary^①，它同样是一个可选的组件，需要在启动时通过 `--with-notary` 参数开启，并且会使用在 Harbor 安装目录下的“`docker-compose.notary.yml`”文件。关于 Notary 的使用方式这里不再展开，有兴趣的读者可以阅读该服务相关文档了解更多信息。

① <https://github.com/docker/notary>

7.5 本章小结

本章介绍了在使用容器集群时经常涉及的基础设施服务。作为规模化之路上的必要保障，性能监控和告警服务为尽早发现集群运营过程中出现的资源瓶颈提供了途径，日志采集服务为大量应用部署后的线上问题定位和业务指标分析奠定数据基础，服务发现是运行于容器集群之上的应用之间相互通信的重要手段，镜像仓库则提供了容器镜像从构建到部署过程中的必要存储能力和分发渠道。

开源社区为这些典型运用场景准备了十分丰富的工具组合，在实际运用中，企业还应当结合自身环境特点和不同工具栈的能力差异进行适当的选型和二次开发，以更好地发挥容器即服务所带来的平台优势。

第 8 章 容器技术新风向

容器技术从来都不是独立发展的，随着这个领域的边界不断扩张，许多新鲜的事物被构造出来。欣欣向荣的 Docker 确实是这股技术浪潮里一朵十分耀眼的浪花，但脑洞大开的探索者们总能发掘出人们意想不到的创意，放眼望去，在开源社区里从来不缺少各种美妙的发明。

本章里将介绍一些和容器技术发展的几个开源项目，作为抛砖引玉。

8.1 安全的集群操作系统: Container Linux

8.1.1 Container Linux 概述

Container Linux^①的前身是 CoreOS 操作系统，后者是最早的一款内置 Docker 容器的操作系统，并以其激进的内核升级策略、平滑的系统升级体验和优秀的集群应用组件而为许多人所知。CoreOS 操作系统诞生于 2013 年 3 月，与 Docker 的诞生相差不到一个月。与许多美国硅谷的创业神话一样，它的三位创始人 Alex Polvi、Michael Marineau 和 Brandon Philips 在美国加利福尼亚州帕洛阿尔托（Palo Alto）市的一个车库里创造了这个操作系统。2018 年 2 月，红帽公司宣布全资收购 CoreOS 公司，后者旗下的 Container Linux 系统和 Etcd 分布式存储等产品都成为了红帽版图的一部分。

与现在的主流 Linux 系统发行版通过补丁包加上发行版周期性的大版本更新不同，Container Linux 将原本常见于应用软件发布的 Alpha、Beta、Stable 升级通道的概念拿到了操作系统发行中，各通道提供不同的发布频率和不同的稳定性保障，用户可以在几种升级通道之间切换，并在后台自动下载和部署新的系统版本，通过一次重启即可完成系统版本

① <https://coreos.com/os/docs/latest>

的切换。其中 Alpha 通道的更新十分频繁，几乎每周或隔周就会推送新的系统版本，让用户在第一时间使用最新的系统组件和内核版本，从而防范网络上的 0day 漏洞攻击。

开发频繁升级的操作系统最大的难点在于如何确保在系统升级过程中不产生兼容性的问题。为此，Container Linux 的开发者做了一个独特的设计：将系统中与系统组件相关的关键目录（包括“/usr”“/sys”“/bin”“/sbin”“/lib”“/lib64”目录）单独挂载成只读的系统分区（准确来说只有“/usr”目录是单独分区，其余目录是为了保持与主流 Linux 目录结构一致性而创建的链接到“/usr”子目录的链接文件），用户无法修改这些目录中的内容，从而确保了核心组件文件不会被意外篡改和破坏。Container Linux 系统的首个正式版本发布日期为 2013 年 7 月 1 日，此后的版本就使用发布日期距离该天的天数作为主版本号，例如版本 247 就是在首次发布之后的第 247 天发布的，即 2014 年 3 月 5 日^①。图 8-1 对比了 RedHat、Ubuntu 和 Container Linux 系统在版本升级时的兼容性。这张图并非表示 RedHat 从 6.x 到 7.x 或 Ubuntu 从 16.10 到 17.04 不能升级，而是官方不能保证升级后系统组件的 100% 兼容性。

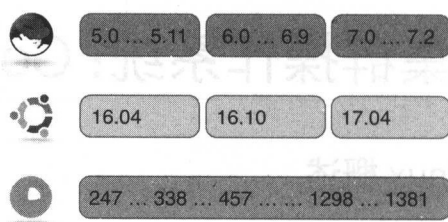


图 8-1 Container Linux 版本升级的连续兼容性

只读系统分区也是一把双刃剑，在带来系统组件完整性的同时，也使得一些需要在安装或运行时修改系统目录的软件无法直接部署在 Container Linux 操作系统。也正是这个原因，使得 Container Linux 在一开始就引入了容器的设计，并将优化容器运行作为系统设计的关键目标。

为了真正将最新的内核和系统组件的安全更新落到实处，Container Linux 还采用了一种大胆的升级策略：自动发现系统新版本，在无人值守的情况下进行自动升级，且升级后若在启动时出现任何问题就自动回滚。这个特性对于大型集群而言尤其有用，整个集群的安全级别取决于其中最脆弱的那个节点，木桶效应是安全领域中不可忽略的防范原则，由于个别主机存在安全漏洞导致整个集群遭受入侵的案例已经屡见不鲜，过去人工应用安全补丁的方式常常会有漏网之鱼，在集群中留下安全隐患。

^① <https://github.com/coreos/manifest/releases/tag/v247.0.0>

系统自动更新为 Container Linux 的设计提出了新的要求，即系统的升级和回滚必须不干扰系统的正常运行，而且启动和回滚的速度必须足够快。为了满足这个需求，Container Linux 首先尽可能地精简了发行版的体积，它的完整 ISO 文件只有不到 200MB，是主流 Linux 发行版的 1/4 左右。其次，Container Linux 系统在安装时，会自动将磁盘多分出两个分区，用来保存系统文件，而设计两个分区的目的在于，当系统运行时总是只挂载其中的一个用于提供使用，而另一个就可以用于后台升级，从而极大地减少了系统升级过程中的等待时间（后台下载完成后，重启自动切换系统分区，整个过程只需十几秒即可完成），如图 8-2 所示。

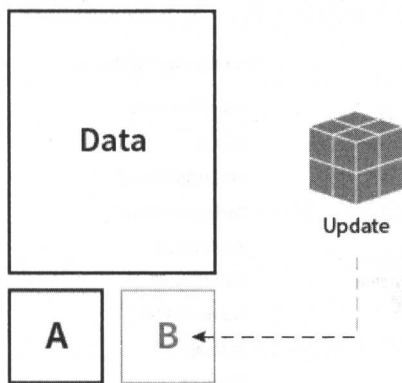


图 8-2 Container Linux 的双系统分区设计

受到技术所限，双系统分区虽然解决了快速升级和出错回滚的问题，更新系统的内核以及关键组件后，依然难免要重启操作系统以便切换内核和组件。因此 Container Linux 将自己定位为专用于集群环境的操作系统，假设集群中所有的服务都是可横向扩展甚至符合 Cloud Native^①以及 12 Factor^②设计的，在这种前提下，集群中任何一个主机的单点重启或故障都不应该导致对外提供的服务中断，为了避免大量节点同时重启，Container Linux 引入了节点重启调度的功能。这种调度功能将系统升级的准备和最终执行重启切换版本的环节分开，使用 Etcd 的分布式锁以及一个专门的 LockSmith 服务，通过集群全局锁确保整个集群所有主机的有序升级。

2016 年 12 月，CoreOS 系统正式重命名为 Container Linux，并设计了单独的 Logo，如图 8-3 所示，使其与 Etcd、Dex、Clair、Rkt 等工具共同成为 CoreOS 公司的开源集群生态一部分。

① <https://pivotal.io/cloud-native>

② https://12factor.net/zh_cn/



图 8-3 Container Linux 的项目 Logo

8.1.2 Container Linux 的部署

作为一款为集群定制的系统，Container Linux 提供了多达十几种公有及私有云平台的定制安装模板，如图 8-4 所示。其中左下角的 ISO 指的是通用的 ISO 文件，它既是系统的 LiveCD，也是快速部署系统到硬盘的工具，为本地主机或虚拟机安装提供便利。

Official Platforms	Community Platforms	
Cloud Providers	Cloud Providers	Other Platforms
Amazon EC2 Container Service	Packet	VMware
Amazon EC2	VEXXHOST Cloud	Libvirt
DigitalOcean	Rackspace Cloud	QEMU
Microsoft Azure	AURO Cloud	Eucalyptus
Google Compute Engine	NIFTY Cloud	Vagrant
Bare Metal	Interoute VDC	CloudStack
Booting with iPXE	Exoscale	VirtualBox
Booting with PXE	Cloud.ca	
Installing to Disk	RimuHosting LaunchtimeVPS	
Other Platforms	Vultr VPS	
OpenStack	Brightbox Cloud	
ISO		

图 8-4 Container Linux 官方收录的支持平台

这里以 VirtualBox 虚拟机和 Vagrant 工具为例，介绍比较适合本地安装和体验的简单部署方式。

首先从网络上下载并安装 VirtualBox 虚拟机软件^①和 Vagrant 虚拟机管理工具^②。然后从 CoreOS 的 GitHub 仓库中下载 Container Linux 的“Vagrant”模板文件，如下所示。

```
$ git clone https://github.com/coreos/coreos-vagrant.git
$ cd coreos-vagrant
```

① <https://www.virtualbox.org>
② <https://www.vagrantup.com>

在这个目录中有三个比较重要的模板文件：“Vagrantfile”“config.rb.sample”和“user-data.sample”。其中第一个文件是虚拟机模板的入口，它对虚拟机的各项参数进行配置，并引用了“config.rb”和“user-data”文件。“config.rb”文件是一个脚本，其中选择性地覆写了一些在 Vagrantfile 中的配置，比如节点数目、规格和 Etcd 服务发现的 Token 等。“user-data”文件则是 Container Linux 启动管理组件 cloud-init 的配置。

Container Linux 的启动管理组件的正式名称是 coreos-cloudinit^①，它是针对 Container Linux 系统的一种开源 cloud-init^②标准实现，后者是一种用于云平台系统初始化的通用开放标准工具，最初在 AWS、OpenStack 等云平台中被用于用户定制系统启动行为的一种机制，它允许在系统启动后立即做一些操作系统的参数和服务配置，如网络配置、用户账户和后台服务等。此外 CoreOS 公司还研发了意图替代 cloud-init 的新一代启动管理组件：Ignition，以提供更高速的系统初始化体验，但由于主流云平台普遍支持 cloud-init 标准，Ignition 的普及还需要一些时间。

cloud-init 配置的表现方式在不同的平台有一些差异。以 Container Linux 的几种安装方式为例，在本地主机或虚拟机安装时，它是一个独立的“user-data”文件。在 OpenStack 等云平台上，它是一个在创建过程中可以填入的 user-data 输入框选项。而在 AWS 中，官方提供的 CloudFormation 脚本中已经包含了这个文件的创建任务，用户只需要在启动时填写一些基本的可配置字段。

在了解完这几个模板文件的作用后，实际的 Container Linux 部署过程非常简单，将仓库里两个后缀是“.sample”的模板文件的后缀去掉，然后在当前目录执行 `vagrant up` 命令。Vagrant 会在默认模板的引导下，自动从网络下载相应版本的镜像文件，然后通过 VirtualBox 启动一个与配置描述匹配的 Container Linux 虚拟机，如下所示。

```
$ vagrant up
Bringing machine 'core-01' up with 'virtualbox' provider ...
... ..
```

等待虚拟机启动完成，使用 `vagrant status` 命令查看这个虚拟机的运行状态，如下所示。

```
$ vagrant status
Current machine states:
core-01                running (virtualbox)
```

① <https://github.com/coreos/coreos-cloudinit>

② <https://launchpad.net/cloud-init>

然后使用 `vagrant ssh` 命令就可以进到 Container Linux 的控制台 Shell 环境了。

8.1.3 Container Linux 的使用

从使用上来说，Container Linux 与普通的 Linux 发行版的主要差异来自于它的只读系统分区，这意味着用户无法直接在操作系统里安装任何传统的系统管理工具。因此最常见的使用 Container Linux 方式是直接在它上面搭建容器集群，将 Container Linux 作为高度自动化的运维环境的一部分，从而既能屏蔽其使用上的差异性，又能充分发挥它在系统自动升级方面的特点，扬长避短。本书第 2 部分所介绍的 4 种容器集群方案都能够使用在 Container Linux 系统上，此外，Container Linux 还与 Google 合作推出了基于 Container Linux 和 Kubernetes 的企业级集群解决方案：Tectonic^①。

同样是因为系统目录只读的缘故，Container Linux 没有提供像 Yum、Apt 这样的包管理工具，如果要使用一些系统没有内置的工具，就得另建一个容器，然后把系统里需要的目录和设备挂载到容器里进行操作，这对于系统的日常运维会比较不方便。为了弥补这个问题，Container Linux 提供了一个 Toolbox 工具来降低运维人员的操作成本。

Toolbox 是一个在 Container Linux 系统上模拟其他主流 Linux 发行版控制台环境的工具，默认模拟 Fedora 运行环境，用户也可以很方便地将它换成其他任意的 Linux 发行版环境。使用 Toolbox 的一个典型场景就是用来运行一些在 Container Linux 里面不容易安装的命令，下面以 `tree` 和 `tcpdump` 为例进行说明。

首先执行 `toolbox` 命令，进入 Toolbox 的运行上下文，这是一个与主机系统共享网络栈、进程上下文、主机名和文件系统目录的 Fedora 发行版容器，如下所示。

```
$ toolbox
Spawning container core-fedora-latest on /var/lib/toolbox/core-fedora-latest.
Press ^] three times within 1s to kill container.
[root@core-01 ~]#
```

使用 `dnf` 安装 `tree` 命令，`dnf` 是较新 Fedora 发行版中替代 `yum` 的一个包管理工具，如下所示。

```
[root@core-01 ~]# dnf install tree
```

然后就可以使用 `tree` 命令了。注意原系统的根目录 “/” 在 Toolbox 环境中已经被自动挂载到 “/media/root/” 目录了，因此在执行相应命令时需要在目标路径前加上

^① <https://coreos.com/tectonic>

“/media/root/”前缀，例如查看主机“/home/core/”目录下的文件结构，如下所示。

```
[root@core-01 ~]# tree /media/root/home/core/
/media/root/home/core/
|-- ...
...
```

建议将“/media/root/”设置为一个环境变量（例如\${ROOT}），以方便使用。

tcpdump 命令的使用方法基本一样，先使用 dnf 安装 tcpdump 命令，然后就可以使用了，如下所示。由于 Toolbox 的运行环境与主机没有网络命令空间隔离，因此能够直接访问到主机的网卡设备。

```
[root@core-01 ~]# dnf install tree
...
[root@core-01 ~]# tcpdump
...
```

另一个需要注意的地方是，在多数主流 Linux 操作系统里，系统“/tmp”目录中的文件是在很长时间不使用后（例如 30 天）才会被清理的，而在 Container Linux 中每次重启都会清空“/tmp”目录的所有文件，不仅如此，同时被清理重建的目录还有“/dev”“/media”和“/run”。如果要在这些目录中长期存放一些固定的文件，应该通过 config-init 在每次系统启动后自动重新创建。

8.2 基于容器的操作系统：RancherOS

8.2.1 RancherOS 概述

RancherOS^①是 Rancher Labs 公司推出的一款基于 Docker 容器而定制设计的 Linux 发行版。它的定制到了什么程度呢，在最初发布后的近一年时间里，Docker 后台服务就是整个 RancherOS 系统 PID 为 1 的根进程，说白了就是将一个 Docker 后台进程托管在 Linux 内核上。与其说它是 Linux 发行版，倒不如说是一个运行在硬件设备上的 Docker 进程，而内核的存在仅仅是为了使用它的设备驱动，让 Docker 有个运行的地方。

RancherOS 系统的运行时结构如图 8-5 所示，系统由两个嵌套的 Docker 进程构成，所

① <http://rancher.com/rancher-os>

有的系统服务，包括管理操作系统设备的 `udev` 服务，管理系统网络的 `netconf`、`dhcpcd` 服务，管理日志的 `rsyslogd` 服务，以及与用户交换使用的 `Shell` 进程，都以容器的形式运行在下层的 `System Docker` 中，而与这些服务一起的还有另一个 `Docker` 服务进程，称为“`User Docker`”，用来以非 `root` 用户的方式运行用户自定义的应用级服务，这样就有效地将用户日常使用的容器与系统容器区分开来，即使用户误删除了 `User Docker` 中的所有容器，也不会影响系统运行。这是典型的 `Docker in Docker` 构造。

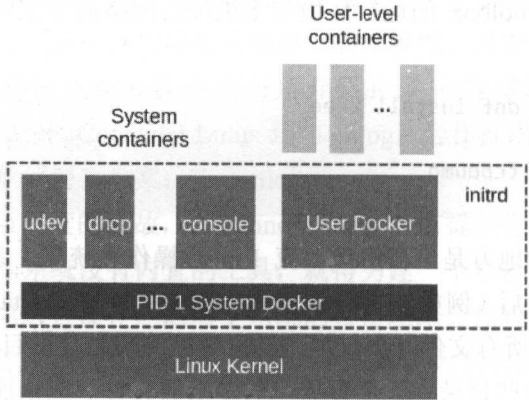


图 8-5 RancherOS 系统结构

从运行方式来说，由于 RancherOS 属于自带内核的操作系统，因此它需要按照虚拟机的方式运行和管理。RancherOS 可以直接运行在 KVM、Xen、VMware 和 VirtualBox 等主流的虚拟机和硬件虚拟化平台之上。而使用 RancherOS 的时候，除了偶尔要和一些工具（例如 `cloud-init` 和 `ros`）打交道，很多时候其实就是在使用 `Docker`：整个操作系统的 `Shell` 是基于 `Docker` 的 `BusyBox` 镜像制作的，要是想换新的还可以拿 `Ubuntu` 或者 `Debian` 的 `Shell` 镜像换掉。若要添加编译内核模块所需的内核头文件，也只需下载指定的 `Docker` 镜像然后启动相应服务。

RancherOS 操作系统的更新十分频繁，紧随 Linux 内核和 `Docker` 版本的快速迭代，因此用户总能在第一时间获得最新的 Linux 内核和 `Docker` 特性。早期版本的 RancherOS 系统 ISO 镜像文件只有 20MB，不过后来新版本 `Docker` 的体积开始“发福”，RancherOS 系统镜像也因此受到牵连，目前 ISO 文件已经增加到 50MB 左右。这里面的主要内容包括：Linux 内核、压缩过的 `Docker` 二进制文件以及一个内置所有系统服务文件的 `Docker` 镜像。这种轻巧的设计也给 RancherOS 系统的在线升级带来很大的便利，第 8.2.4 一节也会对 RancherOS 系统升级操作的细节进行介绍。

8.2.2 部署 RancherOS

RancherOS 与第 5 章介绍的 Rancher 平台都出自 Rancher Labs 公司，两者虽然可以结合使用，但实际上是风马牛不相及的两个独立产品。部署 RancherOS 仅仅指安装操作系统，用户也可以在 RancherOS 上再部署 Rancher，或者部署 SwarmKit、Kubernetes 或 Mesos 等第三方的容器集群解决方案。

既然是操作系统，自然需要部署在虚拟机或者物理机上。RancherOS 提供了在 AWS、GCE、Openstack 等云主机的在线镜像，可以使用在普通 PC 机的 ISO 镜像以及用于 Raspberry PI（树莓派）的 ZIP 镜像。使用云主机镜像当然是最简单的方式，不过，下面从通用性考虑，介绍使用 ISO 镜像安装本地 VirtualBox 或 VMware 虚拟机方法，这种方法适用于普通 PC 物理主机的安装。

首先从 RancherOS 的 GitHub 发布页面^①下载最新的 ISO 镜像文件。使用 VirtualBox 或 VMware 软件创建一个本地的虚拟机，RancherOS 对硬件配置要求不高，即使是只有单核 CPU 和百兆内存的虚拟机，都可以运转得很顺畅。然后将 RancherOS 的 ISO 镜像文件加载到虚拟机的模拟光驱设备里，启动主机，通过光盘引导启动（可能需要在虚拟机配置中选择优先使用光驱启动），即可进入一个运行在内存中的 RancherOS Linux 命令行环境。此时相当于使用 LiveCD 模式运行了一个完整功能的 RancherOS 系统，但在该环境中做的所有修改，包括创建在系统分区的文件都会在重启后丢失。

创建一个名称为“cloud-config.yml”的文件，内容如下所示。

```
#cloud-config
ssh_authorized_keys:
- ssh-rsa <公钥内容>
```

将其中“<公钥内容>”的位置替换为用户自己的 SSH 公用。然后使用 RancherOS 中内置的 `ros` 命令将 RancherOS 系统安装到硬盘上，参数 `-d` 用来表示要部署的磁盘设备，如下所示。

```
$ sudo ros install -c cloud-config.yml -d /dev/xvda
```

这个命令会从网络自动下载与镜像自己版本匹配的 RancherOS 系统安装工具，并在指定的磁盘上进行安装。期间会有两次让用户确认执行的交互环节，直接输入“Y”回车继续，等到安装操作完成后会自动重启系统，此时去掉模拟光驱中的 ISO 文件，就会进入安

^① <https://github.com/rancher/os/releases/>

装在磁盘上并且可以持久化保存运行状态的 RancherOS 环境了。

RancherOS 系统是没有图形界面的，但可以使用 SSH 登录到 Shell 进行操作。此时需要使用 SSH 命令的 `-i` 参数提供私钥文件的位置，用户名是 “rancher”，如下所示。

```
$ ssh -i /path/to/private/key rancher@<ip-address>
```

如果想要离线安装 RancherOS 系统，或是安装与 ISO 镜像自身版本不同的 RancherOS，可以在执行 `ros install` 命令时使用 `-i` 参数直接指定用于安装 RancherOS 系统的 Docker 镜像名称。离线安装时，这个镜像应该提前预加载到 LiveCD RancherOS 的 System Docker 中。

使用 `ros os list` 命令可以查看当前所有可用的 RancherOS 系统安装镜像列表，如下所示。

```
$ sudo ros os list
rancher/os:v1.0.0 remote latest running
rancher/os:v0.9.2 remote available
rancher/os:v0.9.1 remote available
... ..
```

8.2.3 RancherOS 的使用

RancherOS 的使用与普通 Linux 操作系统基本相同，不过由于默认使用的 Shell 环境其实是用一个 Busybox 镜像创建的，其中能够使用的命令相对较少。

使用 `ps` 命令能看到系统中运行的两个核心 Docker 进程 `system-docker` 和 `dockerd`，并且 `system-docker` 的 PID 为 1，如下所示。

```
$ ps aux | grep docker
  1 root      system-docker daemon
 956 root      dockerd
... ..
```

使用 `system-docker` 和 `docker` 命令能够分别对 System Docker 和 User Docker 进行操作，例如查看与系统相关的服务以及提供这些服务的镜像，如下所示。

```
$ sudo system-docker ps
CONTAINER ID  IMAGE                                ...  NAMES
2791ed5a76ff  rancher/os-docker:17.03.1          ...  docker
fa60e8770740  rancher/os-console:v1.0.0          ...  console
95b860706baf  rancher/os-base:v1.0.0             ...  ntp
5a0d14b51d6b  rancher/os-base:v1.0.0             ...  network
```

```
88e13f37c43b rancher/os-base:v1.0.0 ... udev
16c6aca6b16f rancher/os-acpid:v1.0.0 ... acpid
fb96cdba40a4 rancher/os-base:v1.0.0 ... syslog
```

```
$ sudo system-docker images
```

REPOSITORY	TAG	IMAGE ID	...	SIZE
rancher/os-console	v1.0.0	1a9be99012fa	...	27.15 MB
rancher/os-bootstrap	v1.0.0	177a19cd4485	...	27.14 MB
rancher/os-acpid	v1.0.0	54f43e4e2280	...	27.14 MB
rancher/os-base	v1.0.0	20b85226ca27	...	27.14 MB
rancher/os-docker	17.03.1	0481ad94b67a	...	84.87 MB

在 RancherOS 中, 管理系统的服务其实就像管理 System Docker 运行的服务那样。例如在系统中添加一个采集用户容器日志数据的 Filebeat 服务, 如下所示。

```
$ sudo system-docker run -d --name filebeat \
-v /opt/filebeat/filebeat.yml:/filebeat.yml \
-v /var/lib/docker/containers:/var/lib/docker/containers:ro \
prima/filebeat:5.1.1
```

虽然理论上用户可以在 System Docker 中运行任何的服务, 但并不推荐将那些与系统核心功能关系不大的业务服务放到 System Docker 里运行。一方面 System Docker 使用的是版本较稳定的定制 Docker 版本, 不如 User Docker 功能全面, 另一方面, 将日常运维控制在 User Docker 中可以避免由于误操作导致系统关键服务中断的惨剧 (将 System Docker 设计成必须 sudo 运行也是出于类似的安全考虑), 如下所示。

```
$ sudo system-docker version
```

```
Client:
```

```
Version: library-import
```

```
API version: 1.23
```

```
$ docker version
```

```
Client:
```

```
Version: 17.03.1-ce
```

```
API version: 1.27
```

User Docker 的使用与普通 Docker 服务没有什么区别, 如下所示。

```
$ docker run -dt --name nginx nginx
```

```
7bffc3e91708...
```

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	...	PORTS	NAMES
7bffc3e91708	nginx	80/tcp, 443/tcp	nginx

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
------------	-----	----------	---------	------

```
nginx          latest 5766334bdaa0 5 days ago 182 MB
$ docker stop nginx
nginx
$ docker rm nginx
nginx
```

8.2.4 使用 ros 工具管理系统

RancherOS 系统的管理和配置高度依赖于 cloud-config 和 ros。

在介绍 Container Linux 的时候，提到过 Container Linux 的“/run”和“/media”等目录会在每次重启后被还原重建。相比之下，RancherOS 做的有过之而无不及，事实上，RancherOS 只有“/home”“/media”“/mnt”“/opt”这四个目录以及子目录的内容能够在系统重启后被保留，其余目录都会被重建，根目录也无法幸免。通过 mount 命令会发现这四个目录都是单独挂载的，如下所示。

```
$ sudo mount
... ..
/dev/xvda1 on /media type ext4 (rw,relatime,data=ordered)
/dev/xvda1 on /mnt type ext4 (rw,relatime,data=ordered)
/dev/xvda1 on /home type ext4 (rw,relatime,data=ordered)
/dev/xvda1 on /opt type ext4 (rw,relatime,data=ordered)
... ..
```

那么如果希望永久性地修改系统的某些文件，那该怎么办呢？比如为了修改“/etc”目录下的一些系统配置。在 RancherOS 中的解决办法是每次启动的时候都重新生成这些文件。但是每次手动生成显然是不靠谱的，为此就需要借助于 cloud-config。

在通过 ISO 文件部署 RancherOS 的时候其实已经用过一次 cloud-config 所提供的服务，来在启动时每次自动写入 SSH 密钥。当系统部署完成后，那个“cloud-config.yml”文件被存放到了“/var/lib/rancher/conf/”目录中，如下所示。

```
$ sudo ls /var/lib/rancher/conf/
cloud-config.d  cloud-config.yml  metadata
```

值得注意的是这个目录也是单独挂载的，因此用户可以直接修改其中的“cloud-config.yml”文件，它的内容会在下一次重启后生效，如下所示。

```
$ sudo mount
... ..
/dev/xvda1 on /var/lib/rancher/conf type ext4 (rw,relatime,data=ordered)
... ..
```

例如每次重启后在“/etc”目录下面生成一个“demo.conf”配置文件,可以在“cloud-config.yml”文件添加下面这段内容。

```
write_files:
- path: /etc/demo.conf
  permissions: "0755"
  owner: root
  content: |
    <这里的文件的内容>
```

有些系统服务是运行在 System Docker 容器中的,如果要修改这些服务的中的配置文件,只需在 write_files 区域中增加 container 字段就可以了,如下所示。

```
write_files:
- container: ntp
  path: /etc/ntp.conf
  permissions: "0644"
  owner: root
  content: |
    <这里的文件的内容>
```

不过,手动修改这个文件比较容易出现错误,可以通过 RancherOS 提的 ros 工具来更方便地添加属性,如下所示。

```
$ sudo ros config set rancher.network.dns.nameservers \
    ["114.114.114.114",'114.114.115.115']"
```

使用 ros config get 可以获得特定路径的配置值,如下所示。

```
$ sudo ros config get rancher.network.dns.nameservers
- 114.114.114.114
- 114.114.115.115
```

ros config validate 命令可以检查“cloud-config.yml”文件内容是否合法。下面首先创建了一个不符合规范的字段 demo,然后使用 ros config validate 来发现这个问题。

```
$ sudo ros config set demo.rancher "this is invalid"
$ cd /var/lib/rancher/conf/
$ sudo cat cloud-config.yml |sudo ros config validate
> ERRO[0000] demo: Additional property demo is not allowed
```

除了辅助修改启动配置文件,ros 工具提供了 RancherOS 日常管理的许多重要功能,表 8-1 列举了该工具的主要的命令和说明。这些命令大多需要修改操作系统配置,所以通常需要以 root 权限(即 sudo)的方式执行。

表 8-1 Ros 的主要操作命令及其说明

命 令	说 明
sudo ros config	修改操作系统的全局配置
sudo ros service	管理系统的服务组件
sudo ros console	切换系统的控制台环境
sudo ros engine	切换 User Docker 的版本
sudo ros os	切换 RancherOS 操作系统版本
sudo ros tls	配置 User Docker 的 TLS 证书
sudo ros install	安装 RancherOS 系统到硬盘

`ros service` 命令提供了一种管理 RancherOS 系统组件的方法，这些组件实际上也是一些运行在 System Docker 中的容器，它们分别代表了 RancherOS 提供的某种可选系统服务。比如使用 `ros service list` 可以列出当前环境中所有可用的组件，如下所示。

```
$ sudo ros service list
disabled amazon-ecs-agent
disabled crontab
disabled kernel-extras
disabled kernel-headers
disabled kernel-headers-system-docker
disabled open-vm-tools
disabled zfs
```

使用 `ros service enable` 和 `ros service up` 来启动一个系统组件，如下所示。

```
$ sudo ros service enable open-vm-tools
Pulling open-vm-tools (rancher/os-openvmtools:v1.0.0)...
... ..
$ sudo ros service up crontab
> INFO[0000] Project [os]: Starting project
... ..
```

系统组件启动后，在 System Docker 中就会找到提供相应服务的容器，如下所示。

```
$ sudo system-docker ps | grep vmtools
7548ef56241c rancher/os-openvmtools:v1.0.0 ... open-vm-tools
```

RancherOS 允许用户快速切换操作系统的版本（间接改变 System Docker 版本）以及 User Docker 的版本。

先用 `ros os list` 命令列出所有当前可用的 RancherOS 版本，其中状态为“latest running”的那一项就是当前正在使用的系统版本，如下所示。


```
$ sudo ros os list
rancher/os:v1.0.0 remote latest running
rancher/os:v0.9.2 remote available
rancher/os:v0.9.1 remote available
... ..
```

然后用 `ros os upgrade` 将系统直接升级到最新版本，如果要指定升级或降级的目标版本，可以用 `--image` 参数加上期望的镜像版本，如下所示。

```
$ sudo ros os upgrade --image rancher/os:v0.9.2
Upgrading to rancher/os:v0.9.2
... ..
```

若要切换 User Docker，则先用 `ros engine` 命令列出当前所有可用的 Docker 版本，其中状态为 `current` 的那一项为当前使用的版本，如下所示。

```
$ sudo ros engine list
... ..
current docker-17.03.1-ce
disabled docker-17.04.0-ce
```

然后用 `sudo ros engine switch` 命令在不同的 User Docker 版本之间进行切换，如下所示。

```
$ sudo ros engine switch docker-17.04.0-ce
> INFO[0001] Project [os]: Starting project
... ..
$ docker version
Client:
Version:      17.04.0-ce
... ..
```

除了快速切换操作系统的版本和 User Docker 版本，RancherOS 还有一种非常神奇的功能——快速切换控制台环境。RancherOS 默认安装后，用户的 Shell 控制台环境是基于 Busybox 镜像的，其中包含的内置命令较少，甚至没有软件包管理器。不过在 RancherOS 中，控制台环境也只是一个在 System Docker 中运行的容器而已，通过切换镜像，就能让用户快速获得 Alpine、Ubuntu 或 Fedora 等不同 Linux 发行版的控制台体验。下面来演示将操作系统控制台切换成 Ubuntu 镜像的过程。

首先用 `ros console list` 命令列出所有可用的控制台种类，如下所示。

```
$ sudo ros console list
disabled alpine
disabled centos
```

```
disabled debian
current default
disabled fedora
disabled ubuntu
```

然后选择切换控制台为 Ubuntu 类型，如下所示。

```
$ sudo ros console switch ubuntu
Switching consoles will
1. destroy the current console container
2. log you out
3. restart Docker
Continue [y/N]: y
Pulling console (rancher/os-ubuntuconsole:v1.0.0)...
... ..
```

镜像下载完成后，用户会自动登出，SSH 连接断开，此时重新使用 SSH 登录回虚拟机，进入的就是一个与 Ubuntu 发行版几乎一样的 Shell 控制台了，例如可以查看 Ubuntu 的版本，如下所示。

```
$ cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=16.10
DISTRIB_CODENAME=yakkety
DISTRIB_DESCRIPTION="Ubuntu 16.10"
```

可以使用 `apt` 命令安装软件包，如下所示。

```
$ sudo apt update
Get:1 http://archive.ubuntu.com/ubuntu yakkety InRelease [247 kB]
... ..
$ sudo apt install tmux
Reading package lists... Done
Building dependency tree
Reading state information... Done
... ..
```

观察系统的“cloud-config.yml”配置文件，会发现在开头多了一项 `rancher.console` 属性，如下所示。

```
$ sudo head /var/lib/rancher/conf/cloud-config.yml
rancher:
  console: ubuntu
  ssh:
... ..
```

RancherOS 通过高度插件化的设计为用户提供了一种非常灵活的使用体验，可以随时拔插的操作系统组件、简单易用操作系统版本升级和降级，以及平滑的控制台切换足以颠覆许多人对 Linux 系统组件化的认知。

8.3 容器式的虚拟机：Hyper

8.3.1 Hyper 概述

如果说 RancherOS 还只是个运行了 Docker 的 Linux 操作系统的话，Hyper 则是真正地将容器的运行直接搬到了硬件虚拟层上。看到下面这组命令你会想到什么？这是在启动一个 Docker 容器吗？

```
$ hyperctl pull ubuntu:latest
$ hyperctl run -d --name demo ubuntu:latest
$ hyperctl exec demo ls
```

事实上这组命令启动了一个 KVM/Xen 虚拟机(具体哪一种是在 Hyper 安装时确定的)，然后在这个虚拟机里执行了一次 `ls` 命令。那么第一条命令中的那个 `hyperctl pull` 是在做什么呢？脑洞大开的时候到了，这条命令下载了 Docker Hub 仓库里的官方 `ubuntu:latest` 镜像，而之后启动虚拟机使用的正是这个 Docker 镜像！

正如上述例子所演示的那样，Hyper 是一个能够把 Docker 镜像当成虚拟机镜像，将其直接运行在 KVM 或 Xen 的虚拟化硬件资源上的强大工具。它使用了一个高度精简的 Linux 内核，系统的启动时间仅为大约 20ms，达到了与容器同一级别的启动速度。如表 8-2 所展示的那样，Hyper 结合了容器与虚拟机的主要优点。

表 8-2 虚拟机、容器与 Hyper 的比较

对比项目	虚 拟 机	容 器	Hyper
隔离性	强隔离（优势）	较弱的隔离	强隔离
独立内核	是（优势）	否	是
镜像可移植性	平台相关	平台无关（优势）	平台无关
启动速度	数秒至数分钟	毫秒级启动（优势）	毫秒级启动
运行性能（与裸机比较）	有较明显损耗	几乎无损耗（优势）	有轻微损耗
不可变基础设施	不是	是（优势）	是
镜像大小	通过至少几百 MB 到几 GB	可小至几 MB（优势）	可小至几 MB

Hyper 的使用体验实在太像 Docker，它不仅支持从官方的 Docker Hub 或者自建的私有 Docker 仓库获取镜像，还支持将本地的虚拟机镜像推送回 Docker 的镜像仓库中，甚至能够支持推送到那些需要登录验证的仓库。如果将 Hyper 制作成平台化的工具，用户将很难感知其后端运行的究竟是容器还是虚拟机，从而在提升隔离安全性的同时获得容器一样的便捷体验。

当用户启动 Hyper 的操作系统实例时，实际上是启动了一个经过高度精简过的 Linux 系统内核：Hyperkernel。这个内核的启动开销几乎小到可以忽略不计，启动后加载一个 Docker 镜像并运行其中的内容，因此同样做到了每个服务独立使用一个操作系统，又使得这个操作系统中有且只有运行该服务所必需的文件和 library，也做到了每个服务的运行环境高度定制化。虽然 Hyper 依然保留了 Linux “内核态”和“用户态”的切换，但相比普通 Linux 内核，它的执行效率已经要高得多。

在 Hyper 的生态圈中，包含了很多来自 OpenStack 以及 Kubernetes 社区的元素。例如能够将 Hyper 运行在 OpenStack 上的 Nova 驱动插件 Hypernova，将 Hyper 用于与 Kubernetes 进行整合的项目 Hypernetes，以及将 OpenStack 的网络模块 Neutron 用于 Kubernetes 以便于构建 Hyper 集群的 Kubestack 项目等。利用这些现成的平台，站在巨人的肩膀上，Hyper 打造出了一片属于自己的天地。

8.3.2 部署 Hyper

目前 Hyper 的发展分成了两个版本，分别是开源版本^①和 SaaS 版本^②。前者允许用户在自己的 Linux 主机上安装和配置 Hyper 服务，后者则将主机托管在 Hyper 的公有云平台上，用户需按使用的时间和节点的规模付费，价格大约只有同等配置的传统虚拟机的一半。

本小节仅介绍 Hyper 的开源版本在 KVM 虚拟化环境的部署。

KVM 需要虚拟机宿主机的处理器带有虚拟化支持（对于 Intel 处理器来说是 VT-x，对于 AMD 处理器来说是 AMD-V）。可以通过以下命令来检查 CPU 是否支持虚拟化。

```
$ lscpu
```

若 CPU 支持虚拟化，输出结果中会有一行 Virtualization 的信息。另一种使用得比较多的判断方法是执行下面的命令。

```
$ grep -E -o 'vmx|svm' /proc/cpuinfo
```

① <http://hypercontainer.io>

② <https://hyper.sh>

若运行后显示 `vmx` 或 `svm`，则表示 CPU 支持硬件虚拟化。若没有显示任何内容，则表示不支持。注意，有些 CPU 的虚拟化支持是需要 BIOS 中开启的。

确认当前环境的 CPU 支持虚拟化以后，就可以安装 KVM 了。在 Debian/Ubuntu 系统中使用 `apt-get` 命令来安装，如下所示。

```
$ sudo apt-get install qemu-kvm libvirt0 virt-manager
```

在 CentOS/Fedora 系统则使用 `yum` 命令安装，如下所示。

```
$ sudo yum install qemu-kvm libvirt virt-install virt-manager
```

由于 KVM 是一个内核模块，安装完成以后需要重启一次操作系统，重启完成后使用以下命令检查内核是否已经加载 KVM 模块。

```
$ lsmod | grep kvm
```

如果输出了 `kvm_intel` 或者与 KVM 有关的内容则表示 KVM 模块已经就绪，然后就可以使用官方的“install”脚本来一键安装 Hyper 服务了，如下所示。

```
$ curl -sSL https://hypercontainer.io/install | bash
```

这个脚本实际上只是做了操作系统和环境依赖的判断，然后去自动下载所需的安装包。现在官方其实更推荐用户直接下载相应的 DEB 或 RPM 包^①来安装。

最后检查 Hyper 的后台服务 `hyperd` 是否正常启动，如下所示。

```
$ systemctl status hyperd
hyperd.service - hyperd
  Loaded: loaded (/lib/systemd/system/hyperd.service;
         disabled; vendor preset: enabled)
  Active: active (running)
  .....
```

如果看到上面这样的输出，说明 Hyper 服务已经可以使用了。

8.3.3 Hyper 的使用

Hyper 对资源的管理借用了 Kubernetes 中的“Pod”概念，每个 Pod 对应的是 Hyper 的一个虚拟机，本身具有完全独立的网络、进程、用户、主机名等 Namespace 上下文。用户可以在一个 Pod 中启动多个容器，它们将在这个 Pod 中共享大部分的 Namespace，但需要

① https://docs.hypercontainer.io/get_started/install/linux.html

特别注意的是，每个容器的 Mount Namespace 是独立的，这样的设计是必要的，否则各个容器中的文件就要互串在一起了。

虽然 Hyper 启动和管理的是运行在 KVM 或 Xen 等虚拟层上的虚拟机，但它却提供了与 Docker 十分相似的操作命令行工具 hyperctl，其支持的操作及其说明如表 8-3 所示。

表 8-3 Hyper 的主要操作命令及其说明

命 令	说 明
hyperctl run	创建一个 Pod，并用指定镜像启动该 Pod 中的第一个容器
hyperctl list	列出当前 Hyper 服务管理的所有 Pod 或容器
hyperctl rm	删除指定的 Pod 或容器
hyperctl exec	在指定的容器中运行指定命令
hyperctl start	启动一个停止了 Pod 或容器
hyperctl stop	停止一个 Pod 或容器
hyperctl pause	暂停一个运行中的 Pod
hyperctl unpause	恢复一个暂停了的 Pod
hyperctl create	创建一个新的 Pod，或在一个 Pod 里创建一个新的容器
hyperctl attach	将指定容器的标准输入输出重定向到当前控制台
hyperctl build	使用 Dockerfile 创建新的 Hyper 镜像
hyperctl pull	从 Docker 的镜像仓库下载镜像到 Hyper 中存储
hyperctl push	将 Hyper 存储的镜像推送到 Docker 镜像仓库
hyperctl images	列出当前 Hyper 服务管理的所有镜像
hyperctl rmi	删除指定的镜像
hyperctl commit	将指定的容器保存成镜像
hyperctl save	将 Hyper 镜像导出成压缩包
hyperctl load	从指定压缩包导入 Hyper 镜像
hyperctl login	登录指定的 Docker 私有仓库
hyperctl logout	删除指定 Docker 私有仓库的登录信息
hyperctl info	显示与 Hyper 运行环境相关的全局信息

使用 Hyper 时，同样需要先从仓库下载镜像，Hyper 使用与 Docker 相同的镜像地址规则，默认使用 Docker Hub 的 library 组作为镜像来源，如下所示。

```
$ sudo hyperctl pull ubuntu:latest
latest: Pulling from library/ubuntu
...
Status: Downloaded newer image for ubuntu:latest
```

有了镜像以后，就可以启动虚拟机了，hyperctl run 命令在创建虚拟机的同时，还

会在这个虚拟机里使用指定镜像启动一个容器，使用 `-t` 或 `--tty` 参数可以在创建虚拟机后进入相应容器的控制台，如下所示。

```
$ sudo hyperctl run -t --name=ubuntu ubuntu:latest
root@ubuntu:/ # ls
bin dev home lib64 mnt proc run srv tmp var
boot etc lib media opt root sbin sys usr
root@ubuntu:/ # exit
exit
```

类似的，使用 `-d` 或 `--detach` 参数可以让虚拟机在后台启动，如下所示。可以使用 `--help` 参数查看其他可用选项，其中的许多都与 Docker 相似。

```
$ sudo hyperctl run -d --name=ubuntu2 ubuntu:latest
POD id is pod-cRKdMbRdwN
Time to run a POD is 143 ms
```

查看 Pod 列表（虚拟机列表）的命令是 `hyperctl list`，如下所示。

```
$ sudo hyperctl list
POD ID      POD Name  VM name      Status
pod-DieibEsdFL  ubuntu   vm-kIDbsiJebx  succeeded
pod-cRKdMbRdwN  ubuntu2  vm-kIDbsiJebx  running
```

除了通过 `hyperctl` 命令，还可以使用通用的 `virsh` 工具查看当前环境中的 KVM 虚拟机，如下所示。

```
$ sudo virsh list
Id      Name                                State
-----
111     vm-kIDbsiJebx                      running
```

这个命令再次证实了 Hyper 的确创建了一个 KVM 的虚拟机实例。

既然 Hyper 允许一个虚拟机作为 Pod 运行多个容器，能不能一次性创建出完整的 Pod 实例来呢？这就要用到 Hyper 的 Pod 描述文件了，它的作用有点像集群里的容器编排，但并不包括依赖管理等复杂的功能。下面的命令创建了一个简单的 Pod 描述文件，关于 Pod 描述文件的详细配置可参考 Hyper 文档的相应内容^①。

```
$ cat <<EOF >podfile.json
{
  "id": "web-demo",
```

① <https://docs.hypercontainer.io/reference/podfile.html>

```
"hostname": "demo",
"resource": {
  "vcpu": 1,
  "memory": 128
},
"containers": [{
  "image": "nginx:latest",
  "files": [{
    "path": "/var/lib/nginx/conf/",
    "filename": "demo-file",
    "perm": "0755"
  }]
}],
"files": [{
  "name": "demo-file",
  "encoding": "raw",
  "uri": "https://s3.amazonaws.com/bucket/file.conf",
  "content": ""
}],
"volumes": []
}
EOF
```

用带 `-p` 或 `--podfile` 参数的 `hyperctl run` 命令来加载这个描述文件，如下所示。

```
$ sudo .hyperctl run -p podfile.json
```

值得一提的是，实际上 Hyper 并不是唯一想到把容器运行到 KVM 上的项目，类似的还有 Intel 公司的 Clear Containers^① 技术（现在被规划到了 Clear Linux Project 项目下），在第 8.5 一节中会介绍它与 Rkt 结合的运用。

8.4 虚拟机式的容器：LXD

8.4.1 LXD 概述

相比 Hyper 将容器变成虚拟机运行，把容器当作虚拟机用可就不算是新鲜事情了。

① <https://clearlinux.org/documentation/clear-containers.html>

更何况早在 Linux-VServer 和 OpenVZ 时代,所谓的“全虚拟化”工具正是利用 Linux 内核特性把虚拟系统分隔开的。不过本节要专门介绍的 LXD^①来历有点特殊,它与 Docker 的前身 LXC 属于同根同源。

LXD 与 Docker 的关系其实十分微妙,两者本有机会成为正面竞争的对手,却由于各自志向不同,踏上不同的道路。第 1 章里提到过最初版本的 Docker 实际上是 LXC 工具的一层封装,而后者才是最早利用 Linux 内核中的 Namespace 和 CGroup 特性实现了虚拟化运行环境的功臣。随着 Docker 与 LXC 分道扬镳,独立演化成为服务打包和部署的平台工具, LXC 自身也在不断地发展, LXD 正是在这样的背景下诞生的。简单来说, LXD 只是一个守护进程,它为 LXC 容器的管理提供一组 REST API,主要目标是使用 Linux 容器为用户提供一种虚拟机似的体验。

在一个 LXD 典型的容器里,通常会运行一个完整的 Linux 系统,和跑在物理机或者虚拟机上的操作系统几乎一模一样,这些容器基于一个干净的发行版镜像,长期运行。值得指出的是, LXD 并不兼容 Docker 的容器镜像,也没有采用 AUFS 那种层级式的不可变基础设施模式,但是支持对容器运用像虚拟机那样的运行状态快照和还原。Docker 被设计成用来部署和运行用户应用,推崇的是在一个 Docker 容器中只包含一个主要进程的模式,因此它主要提供 PaaS 层的服务,关注于临时的、无状态的、最小化容器上面,通常不会做局部升级或重新配置,而是整个被替换掉。而 LXD 侧重虚拟主机上的应用模式,属于 IaaS 层的服务。因此 LXD 容器会推荐用户使用传统的配置管理工具和部署工具(如 Ansible 或 SaltStack)来进行初始化,并进行增量地更新,用户可以在上面安装很多 Linux 应用,并运行很多的进程,也可以在 LXD 容器里安装 Docker,来运行其他需要的软件。

作为一种虚拟机的 IaaS 运行方案,相比于早期 OpenVZ 等需要修改内核以实现软件虚拟化的做法, LXD 所用的核心技术(Namespace、CGroup、AppArmor、Seccomp、Capabilities 等)都是内置在 Linux 内核中的,可以即装即用,因此它的实施门槛更低,上手起来更加容易。

8.4.2 LXD 的安装和使用

LXD 是 Ubuntu 的母公司 Canonical 所主导的 Container Linuxs 开源技术栈^②的一部分,这个技术栈还包括 PaaS 层的容器实现 LXC、专用的文件系统 LXCFS 以及实现 CGroup 嵌

① <https://linuxcontainers.org/lxd>

② <https://linuxcontainers.org>

套的后台服务 CGManager。因此，LXD 对 Ubuntu 发行版的支持相对较好，在 Ubuntu 中安装 LXD 只需一条命令即可搞定，如下所示。

```
$ sudo apt-get install lxd
```

其他 Linux 发行版可以直接下载 LXD 的二进制包进行安装。

LXD 会自动在系统中创建一个叫作“lxd”的用户组，并自动将原本属于“admin”和“sudo”组的用户自动加入“lxd”用户组，只有在“lxd”组中的用户才能与 LXD 服务通过本地 socket 通信。由于 Linux 用户组的修改需要在用户下一次登录时才会生效，因此应该将当前用户退出重新登录一次，否则执行 lxc 命令时会提示以下错误。

```
error: Get http://unix.socket/1.0: read unix @->/var/lib/lxd/unix.socket: read: connection reset by peer
```

在第一次使用 LXD 前，还需要进行配置初始化。在控制台执行以下命令。

```
$ sudo lxd init
```

该命令会向用户询问一系列的问题，例如使用哪种存储文件系统类型、是否开启远程 TCP 端口、是否配置本地网桥等，按照实际情况回答即可。为了在进行容器热迁移等操作时能够从其他节点远程连接 LXD 服务，建议开启远程 TCP 端口监听。最后屏幕输出“LXD has been successfully configured”，表示可以开始使用 LXD 服务了。未来如果需要修改这些配置，可以直接修改“/etc/default/lxd-bridge”文件。

需要指出的是，LXD 服务的主要职责就是提供 LXC 容器的后台 API、仓库管理、跨节点协调等过去单独依靠 LXC 做不了的事情，而实际的容器功能依然是 LXC 提供的。因此在 LXD 生态中，与容器管理的相关操作使用的还是 lxc 工具。表 8-4 列举了 lxc 工具的主要命令及其说明。

表 8-4 LXC 的主要操作命令及其说明

命 令	说 明
lxc launch	使用指定的镜像创建新容器
lxc list	列出所有由 LXD 管理的容器
lxc stop	停止指定容器
lxc start	启动指定的已停止容器
lxc restart	重启指定的容器
lxc delete	删除指定的容器或快照
lxc exec	在指定的容器中执行 Shell 命令
lxc file	在主机和容器之间拷贝文件，或直接编辑容器中的指定文件

续表

命 令	说 明
<code>lxc snapshot</code>	为容器生成新的快照
<code>lxc restore</code>	将容器还原成指定快照的状态
<code>lxc config</code>	配置 LXC/LXD 的环境参数
<code>lxc copy</code>	使用指定容器或快照克隆出新的容器
<code>lxc move</code>	重命名容器或快照，以及在 LXD 节点之间做容器迁移
<code>lxc image</code>	管理本地的容器镜像
<code>lxc profile</code>	管理预配置文件(包括环境参数、网卡设备等配置的集合)
<code>lxc publish</code>	将容器指定保存成镜像
<code>lxc remote</code>	管理 LXD 镜像仓库
<code>lxc info</code>	显示与 LXD 服务和 LXC 容器相关的信息
<code>lxc version</code>	显示版本信息
<code>lxc help</code>	显示帮助信息

下面简单地演示 LXC 工具（以及 LXD 服务）的基本功能，使用 `lxc launch` 命令创建一个 Ubuntu 16.04 的容器，命名为“c1”，如下所示。

```
$ lxc launch ubuntu:16.04 c1
Creating c1
Starting c1
```

创建一个 CentOS 7 的容器，命名为“c2”，如下所示。

```
$ lxc launch images:centos/7/amd64 c2
Creating c1
Starting c1
```

使用 `lxc list` 命令查看运行的容器列表，如下所示。

```
$ lxc list
+-----+-----+-----+-----+-----+-----+
| NAME | STATE | IPV4 | IPV6 | TYPE | SNAPSHOTS |
+-----+-----+-----+-----+-----+-----+
| c1   | RUNNING | x.x.x.x |      | PERSISTENT | 0 |
+-----+-----+-----+-----+-----+-----+
| c2   | RUNNING | x.x.x.x |      | PERSISTENT | 0 |
+-----+-----+-----+-----+-----+-----+
```

这样看起来，LXD 的容器与 Docker 并没有多少差别。实际是不是这样呢？来看一下在一个典型的 LXD 容器中都运行了哪些东西。使用 `lxc exec` 可以在指定容器里运行命令，如下所示。

```
$ lxc exec c1 ps aux
USER  PID  %CPU  %MEM  ...  COMMAND
Root   1    1.3   0.9   ...  /sbin/init
Root  53    0.2   0.5   ...  /lib/systemd/systemd-journald
Root  54    0.1   0.5   ...  /lib/systemd/systemd-udev
... ..
```

在这个容器中运行了包括 systemd、cron、sshd 等十几个操作系统的核心进程，这些进程构成了一个完整 Linux 运行环境的基础。因此相比 Docker 而言，LXD 启动的容器其实是一个典型的使用全虚拟化技术实现的虚拟机。

使用 `lxc stop` 停止一个容器，如下所示。

```
$ lxc stop c2
$ lxc list
```

NAME	STATE	IPV4	IPV6	TYPE	SNAPSHOTS
c1	RUNNING	x.x.x.x		PERSISTENT	0
c2	STOPPED			PERSISTENT	0

使用 `lxc delete` 删除一个容器，如下所示。

```
$ lxc delete c2
$ lxc list
```

NAME	STATE	IPV4	IPV6	TYPE	SNAPSHOTS
c1	RUNNING	x.x.x.x		PERSISTENT	0

如果删除时指定的容器当前还在运行，应该先停止它，或使用 `--force` 参数强行删除。

接下来，来看一下 LXD 容器的快照功能。下面的例子为 `c1` 容器创建了一个名为 “s1” 的快照，然后向它的 “/root” 目录中拷贝一个文件，再使用 `s1` 快照还原整个容器，如下所示。

```
$ lxc snapshot c1 s1
$ lxc info c1
...
Snapshots:
  s1 (taken at ...) (stateless)
$ lxc file push demo_file c1/root/
$ lxc exec c1 ls /root
demo_file
```

```
$ lxc restore c1 s1
$ lxc exec c1 ls /root
```

可以看到容器被还原后，之前放进去的文件也就不在了。同时值得注意的是，容器可以在线进行快照的创建和还原，并不需先将容器停止，但为了得到内容比较可靠的快照文件，不建议在容器正在繁忙读写文件的时候进行快照操作。

lxc copy 命令使用指定的快照创建新的容器，新拷贝创建出来的容器是处于停止状态的，还需要执行 **lxc start** 将它启动起来，如下所示。

```
$ lxc copy c1/s1 c3
$ lxc list
```

NAME	STATE	IPV4	IPV6	TYPE	SNAPSHOTS
c1	RUNNING	x.x.x.x		PERSISTENT	1
c3	STOPPED			PERSISTENT	0

```
$ lxc start c3
```

lxc move 命令可以用来重命名容器和快照。例如将容器 **c1** 重命名成 “**ubuntu_01**”、快照 **s1** 重命名成 “**snapshot-01**”，如下所示。注意，容器在重命名前必须先停止，而重命名快照则不需要。

```
$ lxc stop c1
$ lxc move c1 ubuntu-01
$ lxc start ubuntu-01
$ lxc move ubuntu-01/s1 ubuntu-01/snapshot-01
```

使用 **lxc image** 命令可以管理本地镜像，例如 **lxc image list** 列出本地已经下载的镜像，如下所示。

```
$ lxc image list
```

ALIAS	FINGERPRINT	PUBLIC	DESCRIPTION	ARCH
	06d46323c907	no	Centos 7 amd64	x86_64
	3e50ba589426	no	ubuntu 16.04 amd64	x86_64

LXD 在安装时会内置三个镜像仓库服务器地址，如下所示。

- “ubuntu”：提供稳定版的 Ubuntu 镜像。

- “ubuntu-daily”：提供 Ubuntu 的每日构建镜像。
- “images”：社区维护的镜像服务器，提供一系列的其他 Linux 发行版。

使用 `lxc remote list` 命令可以管理这些仓库，例如 `lxc remote list` 命令会列出当前 LXD 正在使用的所有镜像仓库，如下所示。

```
$ lxc remote list
```

NAME	URL	PROTOCOL	PUBLIC	STATIC
images	https://...	simplestreams	YES	NO
local (default)	unix://	lxd	NO	YES
ubuntu	https://...	simplestreams	YES	YES
ubuntu-daily	https://...	simplestreams	YES	YES

不过，如果注意观察就会发现，输出的内容中混入了一个看起来不像镜像仓库的地址“unix://”。实际上，LXD 的 Remote 所指的仓库不仅仅是纯粹的容器仓库，还可以是集群中的其他 LXD 节点（此时使用 lxd 协议，而不是 simplestreams），因为 LXD 允许直接从另一节点上拷贝容器和镜像，第 8.4.3 小节中添加 Remote 目标的操作其实就是这种用法。

从以上的例子足以对 LXD 窥斑见豹。LXD 项目的主要负责人 Stéphane Graber 在他的博客^①上写过一些系列文章，详细介绍了 LXD 日常使用的方方面面。国内的 Linux 中国网站也在第一时间将这个系列翻译成了中文^②，有兴趣的读者可以前往阅读。此外，LXD 官方还提供了^③了一个免费虚拟机在线体验 LXD 功能的服务^③，每次可以连续使用 30 分钟。下个小节将向大家重点介绍 LXD 中的一种很值得了解、但在 Stéphane 的博客中写得不够清楚的功能：服务热迁移。

8.4.3 服务热迁移

服务热迁移指的是将服务在不中断当前运行状态的情况下，从一个物理节点移动到另一个物理节点，这是 LXD 的杀手锏级特性。这一功能的实现依赖于 Canonical 公司创造的

① <https://stgraber.org/2016/03/11/lxd-2-0-blog-post-series-012/>

② <https://linux.cn/article-7618-1.html>

③ <https://linuxcontainers.org/lxd/try-it/>

CRIU 技术, CRIU 全称是 Checkpoint/Restart In Userspace (检查点和用户空间重启), 这是一种能够将特定服务的所有运行时信息保存成磁盘文件数据, 然后在另一个地方进行原样还原的技术。正如同它的名字所表述的那样, 它的另一个特别之处在于, 这项技术仅仅通过用户态的代码实现, 无须对 Linux 内核做任何修改。在 Canonical 宣传该功能的时候, 使用了一张活金鱼从一个鱼缸跳到另一个鱼缸的宣传画, 十分形象地展示了这种服务迁移的方式, 如图 8-6 所示。



图 8-6 服务热迁移的宣传画

CRIU 功能需要至少 4.4 以上版本的 Linux 内核, 然后通过以下命令进行安装, 如下所示。

```
$ sudo apt install criu
```

在 CRIU 中的检查点 (Checkpoint) 其实是一种特殊类型的容器快照, 它不仅保存了容器快照创建时刻的磁盘内容, 同时还将当时的内存、CPU 缓存等运行时状态数据全部持久化了下来。它通过一个具有 `--stateful` 参数的 `lxc snapshot` 命令来创建, 如下所示。

```
$ lxc snapshot ubuntu-01 snapshot-02 --stateful
```

还原这种快照的方法和普通快照没有什么区别, 但还原出的新容器将获得快照创建时刻所有程序的运行状态, 如下所示。

```
$ lxc restore snapshot-02 ubuntu-02
```

那么如果把这种带状态的快照发送到另一个节点上去创建容器不就实现了服务“热迁移”了吗? 对的, 不过 LXD 提供了更简便的办法: 用 `lxc move` 直接把容器挪到另一个节点去。

在做这种“乾坤大挪移”之前, 需要先在 LXD 中增加一个目标节点作为 Remote 资源, 使用 `lxc remote add` 命令, 为目标节点起个名字, 加上它的 IP 地址就可以了 (前提是目标主机开启了远程 TCP 端口), 如下所示。


```
$ lxc remote add foo 1.2.3.4
```

此时就可以在执行相关命令时用“<远程主机名>: <容器或镜像名称>”来访问这个主机上的容器和镜像了。比如，可以直接把当前节点上的容器 `c1` 移动到 `foo` 节点，如下所示。

```
$ lxc move c1 foo:c1
```

执行完成后可以分别在两边节点使用 `lxc list` 命令确认一下，就会发现这个容器就这样被不费吹飞之力地搬走了。前一小节说过容器使用 `lxc move` 原地重命名之前要先停止运行，但在使用 `CRIU` 以后，整个迁移过程完全是在线进行的，容器会在新节点上仿佛什么都没发生过似地继续运行其中正在进行的所有任务。

需要提醒的是，`CRIU` 的功能目前还处于测试阶段，在有的主机上经常会发生错误的情况，因此不推荐在比较重要的生产环境中使用。

8.5 容器与虚拟机的统一：Rkt

8.5.1 Rkt 概述

在这个已经十分纷乱的虚拟机和容器社区中，不断有介于容器与虚拟机之间的工具被发明出来，但根据其虚拟化的实现机制最终还是要划归为两者中的其中之一。能不能有一种工具，既能创建、管理容器，又能创建、管理虚拟机，而且还能和社区里的其他集群工具集成使用呢？最早提出这种解决方案的是 `CoreOS` 公司的 `Rkt` 项目。

`Rkt`（发音为 /rɑkɪt/，同英文单词“火箭”）最初是 `CoreOS` 公司设计的一种容器工具，而它当时被创造出来主要是为了制衡变得越发臃肿的 `Docker`。本章在前面介绍 `CoreOS` 公司的 `Container Linux` 系统时提到过，由于这个操作系统的使用高度依赖于容器技术，它所希望的容器工具应该尽可能地纯粹、轻量、安全且稳定。但此时的 `Docker` 发展方向显然与之背道而驰，越来越多的复杂功能被集成到了 `Docker` 的二进制文件里。因此 `CoreOS` 转而开始采用由 `Systemd` 内置的 `Systemd-nspawn` 工具，这个工具虽然轻量却缺少了网络、镜像这些高级功能的支持，`CoreOS` 的工程师们在它之上包装了一层简单的封装。就这样，2014 年 11 月，`Rkt` 的首个版本正式发布，`CoreOS` 同时推出了自己的容器技术规范：AppC Spec^①，

① <https://github.com/AppC/spec/blob/master/SPEC.md>

此规范的内容存在许多与 Docker 不兼容的地方，刚发布就在网络上引起很大的争议。不过由于当时 Docker 的强势垄断早已引起了一些行业巨头的不满，加上 AppC Spec 规范对社区的开放友好性，Google、Red Hat 及 VMware 等公司在 2015 年 5 月的 CoreOS Fest 大会上宣布支持 AppC Spec。眼看形势不妙，Docker 开始做出让步，并在 2015 年 6 月与包括 CoreOS 在内的 35 个企业共同成立 Open Container Initiative^①（开放容器计划组织，简称 OCI），重新确立以 Docker 标准为基础的开放型工业级容器技术规范。

这其中发生了许多精彩或狗血的故事，这里不再逐一细说，当时甚至有人画了如图 8-7 所示的漫画，预示 Docker 将受到 Rkt 的致命打击（图中的“火箭”即指 Rkt 项目）。显然，这样的事情后来根本没有发生，在接下来的几年里，Docker 依然茁壮发展，而 Rkt 则在另一个平行宇宙里自顾自地进化。

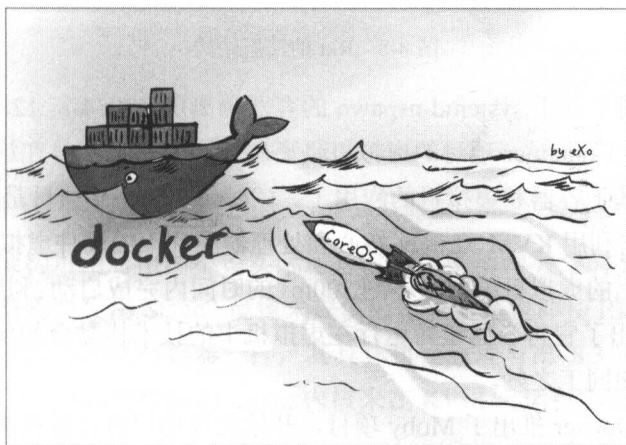


图 8-7 Rkt（原名 Rocket）的漫画

好在功夫不负有心人，2016 年 2 月，Rkt 正式发布 1.0 版本。2016 年 7 月，Kubernetes 的 1.3 版本发布，同时宣布了它的姊妹项目“Rktnetes”，即 Rkt on Kubernetes。为何 Google 的 Kubernetes 工程师们对 Rkt 如此情有独钟呢？这就要从 Rkt 开放式的设计架构说起。

Rkt 将容器的运行划分成如图 8-8 所示的三层结构，其中的 Stage0、Stage1 和 Stage2 可以认为是容器中的服务被创建的三个阶段，每个阶段所用的具体实现可以独立地组合搭配。

其中 Stage0 是用户直接操作的界面层，目前来说实现了这一层的工具就是 Rkt 的命令行。Stage1 是虚拟化层，该层决定最终的应用所采用的隔离方法，这是整个 Rkt 架构的精髓，实现了包括容器和虚拟机在内的不同虚拟化功能无缝接入。Stage2 是服务层，也就是

^① <http://www.opencontainers.org>

用户在容器或虚拟机中运行的实际服务。在通过启动服务时，用户执行 Rkt 命令，并通过参数指定所使用的 Stage1 层类型，然后 Rkt 通过镜像启动 Stage1 对应的服务实例，该实例从 Stage0 中获得用于创建 Stage2 服务所需的镜像以及相关启动参数，并在隔离的环境中启动 Stage2 相应的服务。

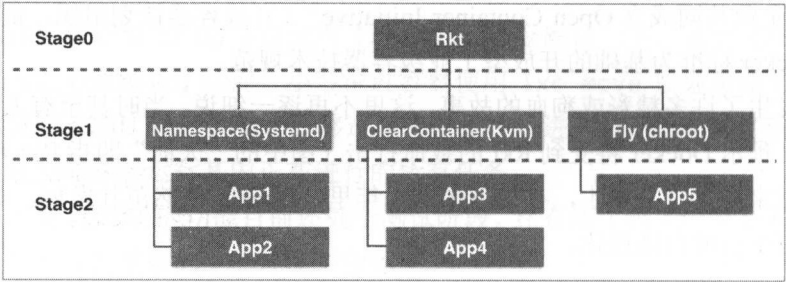


图 8-8 Rkt 的三层结构

最初 Rkt 只实现了基于 systemd-nspawn 的容器隔离层，2014 年 12 月发布的 Rkt 0.8 版本增加了基于 Clear Container 项目的虚拟机隔离层，也就是说，早在这个时候 Rkt 就已经支持同时管理运行基于容器和虚拟机的应用了。Clear Container 项目是 Intel 公司设计的轻量级虚拟化技术，它利用 KVM、Linux 核心和处理器晶片上的硬件虚拟化技术 VT-x，让一个运行在虚拟化层上的虚拟机能够在不到 200ms 的时间内完成启动。利用 Rkt 的封装，Clear Container 充分发挥出了它将容器镜像运行在虚拟机上的技术栈黏合剂角色，这一点与此前介绍的 Hyper 有异曲同工之妙。

2017 年 4 月，Docker 推出了 Moby 项目，开始允许用户自行替换虚拟层的实现。大势所趋，容器和虚拟机技术整合的时代已经不远了。

8.5.2 Rkt 的安装和使用

Rkt 提供了主流操作系统的安装包，可以直接在它的 GitHub 发布页面^①下载。Debian/Ubuntu 系统可以使用 DEB 包进行安装，如下所示。

```
$ wget https://github.com/rkt/rkt/releases/download/v1.25.0/rkt_1.25.0-1_amd64.deb
$ sudo dpkg -i rkt_1.25.0-1_amd64.deb
```

Redhat/CentOS 系统可以使用 RPM 包，如下所示。

^① <https://github.com/rkt/rkt/releases>

```
$ wget https://github.com/rkt/rkt/releases/download/v1.25.0/
rkt-1.25.0-1.x86_64.rpm
$ sudo rpm -ivh rkt-1.25.0-1.x86_64.rpm
```

其他的 Linux 发行版可以下载 tar.gz 格式的通用压缩包，将这个文件解压缩，并进入解压出来的新目录。将这个目录中的 3 个“.aci”文件移动到“/usr/lib/rkt/stage1-images/”目录（需先创建此目录），然后将“init/systemd”子目录中的所有文件移动到系统的“/usr/lib/systemd/system/”目录，将“manpages”子目录中的所有文件放到系统的“/usr/share/man/man1/”目录，接着将“bash_completion/rkt.bash”文件重命名为“rkt”放到“/usr/share/bash-completion/completions/”目录，再将“rkt”可执行文件放到系统 PATH 环境变量指定的目录中，最后用 root 权限执行“scripts/setup-data-dir.sh”脚本，即可完成 Rkt 的安装。

使用 rkt version 命令验证安装可用，如下所示。

```
$ rkt version
rkt Version: 1.25.0
appc Version: 0.8.10
Go Version: go1.7.4
Go OS/Arch: linux/amd64
Features: -TPM +SDJOURNAL
```

从 Rkt 的层级架构可用看出，不论实际运行时使用了哪种类型的 Stage1 虚拟化方法，它所运行的服务本身都是使用 Stage2 镜像来提供的。也就是说即使 Rkt 运行的是虚拟机，它所使用的服务镜像依然和运行容器一样，其具体执行原理与第 8.3 一节中介绍的 Hyper 相似，同样是使用了一个高度精简的 Linux 内核来在 KVM 上运行容器的镜像。

在 Stage2 层，Rkt 支持 AppC 标准的“aci”格式镜像和 Docker 标准的镜像，也同时支持这两种标准相应的镜像仓库协议。先来看一个最简单的例子，使用 Rkt 的默认 Stage1 层，启动一个 AppC 标准镜像的容器，如下所示。

```
$ sudo rkt run coreos.com/etcd:v2.0.12
... ..
gpg key fingerprint is: 18AD 5014 C99E F7E3 BA5F 6CE9 50BD D3E0 FC8A 365E
CoreOS Application Signing Key <security@coreos.com>
Are you sure you want to trust this key (yes/no)?
yes
... ..
gpg key fingerprint is: 8B86 DE38 890D DB72 9186 7B02 5210 BD88 8818 2190
CoreOS ACI Builder <release@coreos.com>
Are you sure you want to trust this key (yes/no)?
yes
... ..
```

连接三次“Ctrl+j”将退出这个 Etcd 容器的运行。

在第一次执行这个命令时，Rkt 会要求用户对信任镜像签名文件的密钥以及镜像发行者的密钥进行确认，这是为了避免第三方篡改本地镜像内容。在 AppC 镜像标准中，所有镜像都需要在发布时使用发布者的私有密钥进行一次签名，并在每次运行时验证签名的匹配性。如果有人在镜像下载后，对本地的镜像进行了修改，由于镜像名称没有变化，用户在执行命令时很难觉察出差异，但由于镜像签名是基于镜像文件哈希编码的，一旦内容变化签名一定会校验失败，因此能有效避免恶意镜像被运行。这是 Rkt 在设计时充分考虑安全性的体现，除此以外，Rkt 对 SELinux 和 AppArmor 等内核安全模块的支持也很好，可以对容器中的应用行为进行细粒度的管控。

Rkt 的容器启动后会保持在前台运行，而无法像 Docker 那样使用 `--detach` 参数将容器放到后台运行，这是一个有意的设计。所有的 Linux 发行版都提供了专门的守护进程执行服务，通常是系统 PID 为 1 的那个服务，例如现在比较常见的 Systemd、早期许多 Linux 采用的 Initd、早期 Ubuntu 系统的 Upstart 以及 RancherOS 的 System Docker 等，这些守护进程服务都有完善的日志管理、自启动管理等配套功能。而 Docker 虽然提供了方便的 `--detach` 参数，但那是以另立山头、使用自己单独的后台管理进程为代价的，这在 Rkt 看来是一种复杂且封闭的重造轮子的行为。

以使用 Systemd 的系统为例，如果希望将服务放到后台运行，可用 `systemd-run` 工具来运行 Rkt，如下所示。

```
$ sudo systemd-run --slice=machine rkt run coreos.com/etcd:v2.0.12
Running as unit run-r4b600a04471c44faa625438fd415cd14.service.
```

命令执行完，控制台将立即返回，并打印自动创建出的服务名称。使用 `systemctl status` 命令可以查看服务的运行状态，如下所示。

```
$ systemctl status run-r4b600a04471c44faa625438fd415cd14.service
●run-r4b600a04471c44faa625438fd415cd14.service -
/usr/bin/rkt run --volume data-dir,kind=host,source=/tmp/data coreos.com/e
Loaded: loaded
Transient: yes
Drop-In: /run/systemd/system/run-r4b600a04471c44faa625438fd415cd14.service.d
└─50-Description.conf, 50-ExecStart.conf, 50-Slice.conf
Active: active (running)
... ..
```

若要停止并删除它，应该使用 Systemd 服务管理的标准方法，如下所示。

```
$ sudo systemctl stop run-r4b600a04471c44faa625438fd415cd14.service
```



```
$ sudo rm -fr /run/systemd/system/run-r4b600a04471c44faa625438fd415cd14.service.d
$ sudo systemctl daemon-reload
```

Rkt 可以使用 Docker 的镜像仓库，只需在镜像名称前面加上 “docker://”，如下所示。

```
$ sudo rkt --insecure-options=image run docker://redis
... ..
```

注意命令中的 `--insecure-options` 参数，它会让 Rkt 在使用镜像时忽略对签名的检查，这是因为 Docker 标准的镜像是没有提供签名文件的，若不用这个参数，Rkt 会拒绝运行这些镜像。这个 Redis 服务启动后同样地会保持在前台，按三次 “Ctrl+J” 可将其退出。

上个小节介绍过 Rkt 的 Stage1 层功能同样是使用镜像来提供的，在启动服务时通过 `--stage1-name` 参数就可以更换使用的 Stage1 实现。CoreOS 公司提供了三个官方支持的 Stage1 镜像，也就是在安装包中的那三个 “.aci” 文件。下面三个命令分别展示了这三种不同 Stage1 层启动 Etcd 的服务，如下所示。

```
$ sudo rkt run --stage1-name=coreos.com/rkt/stage1-coreos:1.25.0 \
  --volume data-dir,kind=host,source=/tmp/data coreos.com/etcd:v3.1.5

$ sudo rkt run --stage1-name=coreos.com/rkt/stage1-fly:1.25.0 \
  --volume data-dir,kind=host,source=/tmp/data coreos.com/etcd:v3.1.5

$ sudo rkt run --stage1-name=coreos.com/rkt/stage1-kvm:1.25.0 \
  --volume data-dir,kind=host,source=/tmp/data coreos.com/etcd:v3.1.5
```

其中第一个命令的 `--stage1-name` 参数可以省略，默认使用的就是这个 Stage1 镜像。第二个命令使用的 `stage1-fly` 镜像实现了最轻量的服务运行环境，它基于 Linux 的 `chroot`，也就是说仅仅在磁盘空间上进行了隔离，服务本身与主机使用完全相同的资源 Namespace。第三个命令使用的 `stage1-kvm` 就是基于 KVM 的虚拟化实现了，使用这种虚拟化的前提同样是当前环境支持 CPU 虚拟化，且已经安装 KVM 的内核组件以及 QEMU 相关服务。

为了统一虚拟机和容器虚拟化运行的服务，Rkt 中将通过 `rkt run` 创建的服务单元称为 “Pod”。实际上，它与 Kubernetes 中的 “Pod” 在概念上是一致的。`rkt run` 命令本来就可以一次性运行多个容器，这些容器会共享相同的 Namespace（Systemd-nspawn 作为 Stage1 的情况）或虚拟机（KVM 作为 Stage1 的情况）。下面这个命令创建了包含两个容器的 Pod。

```
$ sudo rkt run demo/app1 demo/app2
```

通过 `rkt list` 命令可以看到，这两个容器共用了 Pod ID 和 IP 地址，如下所示。

```
$ sudo rkt list
  UID  APP  IMAGE NAME          STATE  ... Network
  7f1... app1 demo/app1:latest  running  default:ip4=172.16.28.4
      app2 demo/app2:latest
```

Rkt 工具的可用命令比较多，表 8-5 列举了其中最主要的部分。

表 8-5 Rkt 的主要操作命令及其说明

命 令	说 明
rkt run	创建并启动一个 Pod
rkt prepare	预备运行指定 Pod 所需的资源和镜像
rkt run-prepared	启动一个已预备好的 Pod
rkt list	列出所有的 Pod
rkt status	检查指定 Pod 的运行状态
rkt enter	进入指定 Pod 的 Namespace 空间
rkt stop	停止指定的 Pod
rkt rm	删除指定的 Pod 使用的本地资源，并将 Pod 标记为待删除
rkt gc	清理所有标记为待删除的 Pod
rkt export	将指定容器导出成 ACI 镜像文件
rkt cat-manifest	显示指定 Pod 的详细描述信息
rkt image list	列出所有的本地仓库镜像
rkt image render	将指定本地镜像及它依赖的镜像合并后展开到指定目录
rkt image extract	将指定的本地镜像内容（不包括依赖镜像）展开到指定目录
rkt image export	将指定的本地镜像导出成 ACI 镜像文件
rkt image gc	清理本地镜像仓库长时间没有使用过的镜像
rkt image rm	将指定镜像从本地仓库中移除
rkt image cat-manifest	显示指定镜像的详细描述信息
rkt trust	信任指定的镜像签名密钥
rkt fetch	获取指定镜像并保存到本地仓库
rkt metadata-service	启动 Rkt Metadata 后台服务
rkt api-service	启动 Rkt API 后台服务
rkt config	显示 Rkt 的 Stage0 和 Stage1 配置

Rkt 的 Pod 生命周期分成“预备（Prepare）”“运行（Run）”“结束（Exited）”“待回收（Garbage）”四个主要状态，以及“起始（Embryo）”和“预备完成（Prepared）”两个辅助状态，它们之间的转化关系如图 8-9 所示。

当执行 `rkt run` 时，Rkt 的 Pod 实际上经历了从初始、预备到启动运行的几个阶段，直接进入运行状态，这也是最常用的一种 Pod 使用方式。但按照 Rkt 的 Pod 生命周期，`rkt`

run 操作实际上可以分解成 `rkt prepare` 和 `rkt run-prepared` 两步。

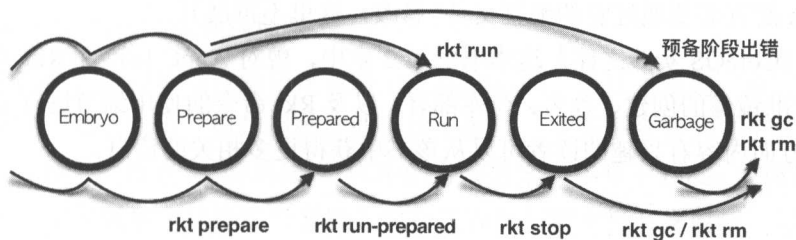


图 8-9 Rkt Pod 的状态切换

`rkt prepare` 操作会检查并下载好运行 Pod 所需的镜像和存储卷，创建 Pod 的 UUID，但不会启动 Pod，因此也不会创建网络 IP，占用 CPU、内存。而 `rkt run-prepared` 则会启动一个已经准备好的 Pod。在什么情况下需要把准备和启动两个步骤分开进行呢？其实，Rkt 的 Pod 启动速度是非常快的，但检查镜像和存储等所需准备工作的时间可能会长得多，特别是比如所需的镜像还未在本地保存，或是所需的存储资源需要临时分配的情况。因此在服务批量升级或者其他必须快速完成切换的场景下，在正式启动前，预先准备好所有资源可能是必要的操作。

另外，十分值得大家注意的一点是，Rkt 的 Pod 生命周期是单向的。在 Rkt 中只有 `rkt stop` 命令，而没有 `rkt start` 或 `rkt restart`，这同样是一个故意的设计。就像是一个普通的服务进程运行结束了，下一次再启动就是一个新的新进程（不同的 PID），在 Rkt 的世界观中，不论是直接运行在系统的普通进程、运行在 Pod 里的进程还是运行在虚拟机里的进程，都具有一致的行为模式。因此在 Systemd 这样的服务管理器里就不必关心“启动”到底应该是重建新的服务实例还是恢复之前的服务实例（这是在用其他服务管理软件管理 Docker 容器时很麻烦的一件事）。

除了与容器或镜像相关的命令，在 Rkt 中还有两个后台服务：Metadata 服务和 API 服务。它们分别使用 `rkt metadata-service` 和 `rkt api-service` 命令启动，这两个命令都会保持在前台运行，所以建议将 Systemd 作为系统服务管理。Metadata 服务能够为所在节点上的 Rkt 容器提供关于 Pod 的自身信息，包括当前 Pod 自己的 UUID、IP 地址和其他属性（这部分在文档写得比较含糊，可以直接从代码的单元测试^①里找到），需要在 Pod 启动的 `rkt run` 命令加上 `--mds-register` 参数，然后通过运行上下文中自动注入的“`AC_METADATA_URL`”环境变量获得 Metadata 服务的地址。API 服务对外界提供了获取当前节点上所运行的 Pod 信息的接口（gRPC 协议），它目前还不提供对 Pod 进行创建、修改和删除操作的 API，因此这个服务实际上很少会被用到。

① https://github.com/rkt/rkt/blob/master/rkt/metadata_service_test.go

在表格最后的 `rkt config` 命令仅仅是用来显示当前 Rkt 有关 Stage0 和 Stage1 配置的，而实际的 Rkt 配置需要通过它的配置文件^①修改，这里不再展开。

笔者在《CoreOS 实践之路》这本书的第 2 章中，曾对 AppC 标准和 Rkt 工具的发展过程，AppC 标准镜像的创建、签名、仓库部署，以及 Rkt 命令的使用细节进行了比较详细的描述。对这方面内容有兴趣的读者可以从该书中获得更多相关的信息。

8.6 企业级定制容器：Pouch

8.6.1 Pouch 概述^②

Pouch[®]本意为育儿袋，它是一款阿里巴巴公司开源的企业级容器产品，其前身可以追溯到 2011 年阿里巴巴基于 LXC 研发的容器技术“t4”。随着时间的推移，两年后 Docker 横空出世，在镜像技术层面解决了困扰行业多年的“软件封装”问题。此时正值阿里巴巴业务量的爆炸式增长时期，阿里基础设施团队体验到大规模运用容器带来成本降低的甜头，开始在容器技术领域加大投入，很快就在自身容器技术 t4 的基础上，吸收 Docker 中的镜像技术并加以改进，将其演变为另一款内部产品“AliDocker”持续孵化。最终于 2017 年底，AliDocker 正式更名为 Pouch，在同年的云栖大会上宣布开源。

所谓实践出真知，Pouch 历经 7 年打磨，早已在阿里集团内部占有重要地位，其应用场景覆盖了阿里绝大部分的事业部，涉及电商应用、数据库、实时搜索、大数据、流计算等众多领域。近年来，每年的双 11 购物节都可以算是一次“超级工程”，巨大流量的背后正是庞大的 Pouch 容器集群提供了运行环境支撑^④。而在 2017 年的双 11 中，当日创历史新高除了 1682 亿的交易额，还有峰值达到百万级的容器调度规模和 100% 的在线业务服务 Pouch 化。

由于 Pouch 直接源于阿里的企业级场景而非社区，因此在容器效果、技术实现等方面，还是与 Docker 体系存在一些显著的差异。相比之下，Pouch 拥有隔离性强、镜像分发效率

① <https://coreos.com/rkt/docs/latest/configuration.html>

② 此小节的内容部分来源于文章 <http://www.infoq.com/cn/articles/alibaba-pouch>，已征得原作者同意。

③ <https://github.com/alibaba/pouch>

④ <http://jm.taobao.org/2017/04/06/20170406/>

高、业务侵入性低、可移植性好等特点，更易于在超大规模的数据中心使用。

1. Pouch 对内核隔离性的增强

隔离性是在企业级基础设施云化道路上无法回避的一个技术难题。隔离性强，意味着对服务影响低、易于商用，但却很难在业务线上推广，尤其是容器的安全性问题，更是环境隔离里十分棘手的难点。众所周知，社区中的容器方案大多基于 Linux 内核提供的 CGroup 和 Namespace 实现隔离，这样的轻量级方案存在容器与宿主机共享内核，以及内核级资源隔离不够彻底的弊端。根据这种现状，Pouch 采取了以下四个方面的措施来增强容器的安全：

- 用户态增强容器的隔离维度，比如网络带宽、磁盘使用量等。
- 实现类似 Hyper 的、基于硬件虚拟化的容器，通过创建新内核实现容器隔离。
- 率先支持 LXCFS，保持容器内部资源视角与容器的资源限制结果一致。
- 给内核添加补丁，修复容器的资源可见性问题，特别是 CGroup 方面的 Bug。

对于容器安全的讨论，在社区里已经持续了相当长的时间，Pouch 项目的出现为这一领域再次注入新的血液。同时，阿里巴巴也在计划开源“阿里内核”，将多年来阿里对 Linux 内核的增强回馈业界，减少容器技术在内核级别上的安全隐患。

2. P2P 镜像分发

随着容器规模的持续增长，容器镜像的分发会成为限制容器服务创建速度的瓶颈。举个简单的例子，假设数据中心有一万台物理节点，每个节点同时向镜像仓库发起镜像下载，即使每个镜像平均大小为 100MB，此时镜像仓库所在机器需要的网络负载和 CPU 压力都将是天文数字。在阿里量级的容器集群中，传统的镜像分发方式效率令人抓狂。

在此背景下，阿里巴巴自研的基于 P2P 技术的镜像分发工具“Dragonfly”（蜻蜓）应运而生（目前该项目同样已在 Github 开源^①）。蜻蜓采用智能路由算法实现运行节点间的去中心化镜像分发，能消除大规模文件分发场景下的中心节点传输瓶颈，解决分发耗时、成功率低、带宽浪费等问题，从而大幅提升数据预热、快速部署、大规模镜像同步的业务能力。图 8-10 展示了蜻蜓与 Pouch 容器配合进行镜像传输的大致原理。值得注意的是，镜像的数据是以镜像层文件为单位同步的，符合容器镜像层的复用原理，有助于最小化数据传输量。

^① <https://github.com/alibaba/dragonfly>

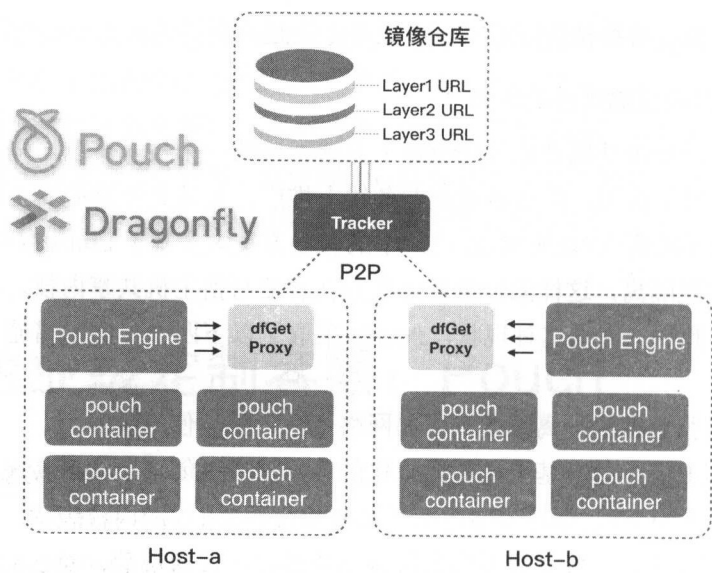


图 8-10 蜻蜓与 Pouch 协同工作的原理示意图

3. 富容器技术

阿里巴巴集团内部囊括了各式各样的业务场景，几乎每种场景都对 Pouch 有着自己的要求。如果使用 Docker 社区推崇的“单容器单进程”方案，在业务部门推行容器化会存在令人难以置信的阻力。与许多大型企业一样，阿里巴巴的基础设施团队承担着巨大的底座作用，为业务的稳定运行撑起一片天。在业务高速发展的时期，想让业务部门放下手里忙不完的活，去配合基础技术演进的需求，几乎是不可能的事。因此，一种对应用开发、应用运维来说都是最低侵入性的容器技术，才有可能大规模地迅速铺开。否则在容器化过程中，很容易面临既得不到业务方支持，还要投入大量人力帮助业务方解决技术债、实施“非标准化”运维的尴尬境地。

为此，Pouch 在设计时致力于减少项目迁移时可能对业务产品运行形态的任何形式侵入。也正是基于这一点，Pouch 能够在阿里集团内部实现业务服务 100% 容器化的目标，与之相关的关键技术手段被称为“富容器”。

从本质上来说，富容器的原理与第 8.3 一节介绍的 Hyper 有异曲同工之处，但设计思想则完全不同。富容器技术的目的是在 Linux 内核上创建与虚拟机体验完全一致的容器环境。譬如，在容器中包含完整的 Init 启动进程、Crontab 定时器进程，以及业务应用需要的任何系统服务。当然这也印证了 Pouch 为什么可以做到对应用没有“侵入性”。具体来说，Pouch 会将容器的执行入口定义为 Systemd，而在内核态，Pouch 引入了 CGroup Namespace

这一最新的内核特性（可参见第 1.1.2 小节），从而满足了 Systemd 在富容器模式下的隔离性。从企业运维流程来看，富容器可以在应用的 Entrypoint 启动之前做一些事情，比如与安全相关的事情、与运维相关的 Agent 拉起。倘若将这些需要统一做的事情放到用户的启动脚本或镜像中，就对用户的应用产生了侵入性，而富容器可以透明地处理这样的任务。

4. Pouch 的内核兼容性

容器技术的井喷式发展使得不少走在技术前沿的企业享受技术红利。然而，“长尾效应”注定技术演进存在漫长周期。凡是规模达到一定量级的数据中心，必然会积存老旧型号的硬件资源，如何利用与处理这些遗留设备是一个大问题，Pouch 也在规模化进程中遇到了这样的情况。

无论是不同型号的机器，还是从 2.6.32 到 3.10+ 的 Linux 内核，异构现象无法避免。倘若要使所有应用运行于容器之中，使用的容器就必须支持所有的内核版本，而现有的容器技术支持的 Linux 内核都在 3.10 以上。不过万幸的是，在技术层面，对 2.6.32 等老版本内核而言，Namespace 的支持仅仅缺失 User Namespace，其他 Namespace 以及常用的 CGroup 子系统均存在。但当时还都没有“/proc/self/ns”等用来记录 Namespace 的辅助文件，setns 等系统调用也需要在高版本内核中才能支持。对此，Pouch 通过特殊的技术手段，绕过了某些系统调用，从而实现老版本内核的容器支持。从另一个角度而言，富容器技术也在很大程度上对传统的运维系统、监控系统等实现了适配，保障 Pouch 在用户使用习惯上同样对原有的设备管理者有良好的兼容性。

从 Pouch 的这些技术特点里，我们不难发现 Pouch 的定位是一款针对传统 IT 架构迅速容器化，企业级大规模业务部署，以及安全隔离要求高、稳定性要求高的金融业务等场景而定制的容器产品。相比 Docker 彻底改造服务交付方式的情怀，Pouch 则显得更加中庸实用，以一种温和而友好的姿态帮助业务完成容器化转型。也恰恰是这种不那么激进的风格，使得 Pouch 能够在阿里这样复杂的企业级场景中铺展得踏踏实实、游刃有余。

8.6.2 Pouch 的开源生态

虽然阿里 Pouch 的设计思路清奇、优势独特，但若是将内部版本的 Pouch 直接搬到开源社区，那几乎是一件没人能够玩转的玩意。在多年的曲折演进中，内部版本 Pouch 在服务业务的同时，存在与阿里底层基础设施、各类特殊业务场景错综复杂的耦合情况。其中许多耦合的内容对于社区而言并没有什么通用性，却会引入更多麻烦的问题。正如 Google 基于内部的 Brog 开源设计 Kubernetes 一样，Pouch 开源的第一要务其实是解耦内部依赖，把最核心

的、对社区有同样巨大价值的部分梳理出来。因此，当前开源版本的 Pouch 并非直接照搬原有的 Pouch 代码，更像是以之为基础的一次大规模重构。同时，Pouch 团队也表示，开源版本 Pouch 将逐渐替换集团原本的内部 Pouch，最终达成 Pouch 内外版本一致的目标。

在组件架构上，Pouch 采用与 Docker 相似的客户端/服务端架构形式，且在接口层面上兼容，其客户端 CLI 工具的命令和参数也与 Docker 相仿，因此未来甚至能够实现直接通过 Docker CLI 操作 Pouch 容器。Pouch 的容器运行时采用业界一致认可的 Containerd 标准，并且在主流的 runC、runV 等实现之外，自研了 runxc 容器运行时，用于提供兼容低版本 Linux 内核的容器实现。Pouch 通过 gRPC 协议完成客户端与服务端通信，服务端的内部采取组件化的设计理念，抽离出 System Manager、Container Manager、Image Manager、Network Manager、Volume Manager 等多个模块（与 Kubernetes 的 Controller Manager 服务的设计思路类似），提供统一化的对象管理方案。

在开源生态圈的建设上，Pouch 正在积极融入 CNCF 基金会，并已经有了自己的计划蓝图，如图 8-11 所示。从这个生态圈中可以看到几条比较明显的主线：各类主流容器编排系统（Kubernetes、Swarm、Sigma 等），其中 Sigma 是阿里自研的一套容器编排调度系统；各类主流的容器运行时实现（runC、runV、runxc）；以及主流网络、存储模型（CNI、CSI）和蜻蜓等辅助系统。Pouch 自身则作为整个生态系统的容器管理引擎，与 Docker 等主流容器工具遵循相同的容器引擎标准。

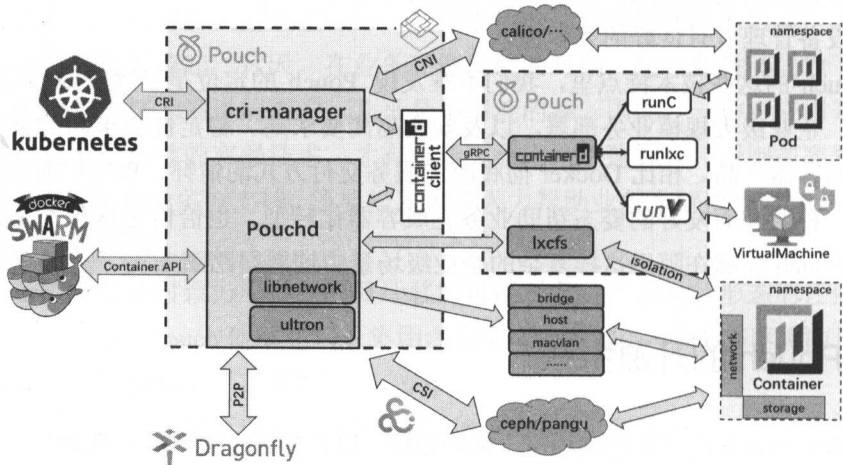


图 8-11 Pouch 的开源生态圈

在容器编排系统方面，除了原有的 Sigma 调度系统，Pouch 将优先支持 Kubernetes 和 Swarm。为实现这一点，Pouch 在设计过程中充分考虑了 Docker 等社区工具模型的一致性，

兼容 CRI 模型和 Container API，为编排方案的融合奠定了基础。

在容器运行时方面，Pouch 目前已经支持 Containerd 标准的 v 1.0.0 版本^①。这个版本与此前的 0.x 版本有较大的变化，使得 Pouch 能够采用几乎相同的接口切换 runC、runV 等开源容器运行时。Pouch 的容器运行时设计如图 8-12 所示。

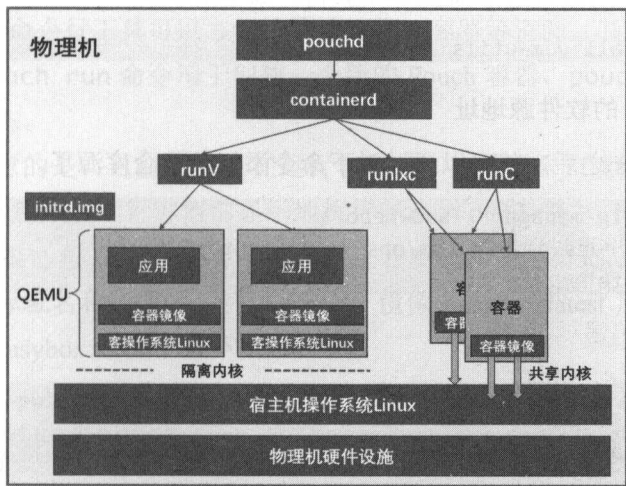


图 8-12 Pouch 的容器运行时接口

在存储和网络方面，Pouch 将实现 CNI 和 CSI 接口，从而充分利用社区已有的资源，不必重造接入分布式存储和跨节点网络的轮子。Pouch 在监控方面支持主流的 Prometheus 标准数据格式。阿里还在持续输出其他的 Pouch 周边产品，之前介绍过的蜻蜓就是一个例子。此外，作为 Pouch 项目的合作伙伴，浙江大学 SEL 实验室也在 Pouch 的 Kubernetes 支持、增强容器运行时和其他周边领域贡献了许多力量。可以预见不远的未来，在社区的积极推动下，Pouch 的开源生态必将日益完备。

8.6.3 体验 Pouch

目前 Pouch 提供了两种通用的部署方式。第一种是软件包安装，第二种是源码安装。对于红帽系（RHEL、CentOS、Fedora）操作系统的用户，Pouch 官方建议使用软件包安装的方式。Linux 操作系统的用户可以选择通过源码构建安装。

下面以安装 RPM 软件包的方式举例。Pouch 的 RPM 软件包托管在阿里云的包仓库中，

^① <https://github.com/containerd/containerd/releases/tag/v1.0.0>

用户只需添加这个软件仓库源就可以使用 Yum 等标准包管理工具来安装和更新 Pouch。

1. 安装 yum-utils

在安装 Pouch 之前，首先要确保系统中已有 `yum-config-manager` 命令，它包含在 `yum-utils` 软件包里，可以通过 `yum` 命令来获得，如下所示。

```
$ sudo yum install yum-utils
```

2. 设置 Pouch 的软件源地址

安装完 `yum-utils` 之后，就可以通过以下命令添加软件仓库源了。

```
$ sudo yum-config-manager --add-repo \
    http://mirrors.aliyun.com/opsx/opsx-centos7.repo
$ sudo yum update
```

3. 安装 Pouch

通过以下命令就可以安装最新版本的 Pouch。

```
$ sudo yum install pouch
```

第一次安装 Pouch 时会提示用户校验安装包的 GPG 密钥指纹，如下所示。

```
Retrieving key from http://mirrors.aliyun.com/opsx/pouch/linux/centos/gpg
Importing GPG key 0xCFE5283C:
  Userid: "Pouch Packages RPM Signing Key <pouch-dev@list.alibaba-inc.com>"
  Fingerprint: xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
  From: http://mirrors.aliyun.com/opsx/pouch/linux/centos/gpg
Is this ok [y/N]: y
```

这是 Yum 对软件包来源的安全签名机制，输入 `y`，然后回车即可。

4. 启动 Pouch 服务

Pouch 安装时会自动将后台服务 `pouchd` 配置为 `Systemd` 管理的系统服务，可以使用 `systemctl` 命令来启动它，如下所示。

```
$ sudo systemctl start pouch
```

启动完毕后用 `pouch version` 命令检查后台服务是否正常运行，如下所示。

```
$ sudo pouch version
KernelVersion:
Os:            linux
Version:       0.1.0
APIVersion:    1.24
```



```
Arch:          amd64
BuildTime:     .....
GitCommit:
GoVersion:     go1.9.1
```

5. 管理容器

使用 `pouch` 命令行工具可以方便地管理镜像、容器等资源，其命令参数与 `Docker` 十分相似，例如 `pouch run` 命令用于创建一个新的 `Pouch` 容器，`pouch ps` 命令用于查看所有容器的运行状态。

这里需要注意的是，目前 `Pouch` 在运行容器时，如果所需的镜像不存在，默认不会自动下载镜像。因此应该在运行容器前显示地执行 `pouch pull` 命令，拉取所需的镜像。`Pouch` 镜像遵循 `OCI` 标准，与 `Docker` 镜像通用，不过若要使用来自 `Docker` 官方仓库的镜像，必须写出完整的仓库域名和镜像组。例如 `Docker` 镜像 `busybox:latest` 应该写为完整仓库地址 `docker.io/library/busybox:latest`，如下所示。

```
$ sudo pouch pull busybox:latest
busybox:latest: resolving |-----|
elapsed: 0.0 s total: 0.0 B (0.0 B/s)
Error: failed to display progress: failed to pull image: object required

$ sudo pouch pull docker.io/library/busybox:latest
docker.io/library/busybox:latest: resolved |+++++++|
index-sha256:1669a6a....a748db: done |+++++++|
manifest-sha256:4cee....ff352d: done |+++++++|
layer-sha256:5731016....3bf566: done |+++++++|
config-sha256:5b0d59....5bd2c3: done |+++++++|
elapsed: 1.9 s total: 710.7 (373.7 KiB/s)
```

启动一个 `Pouch` 容器，然后查看当前节点上的所有容器，如下所示。

```
$ sudo pouch run --name test docker.io/library/busybox:latest echo "hi"
hi
$ sudo pouch ps -a
Name   ID       Status   Created   Image                                     Runtime
test   5119b4   stopped   ...       docker.io/library/busybox:latest         runc
```

即使在使用方式上十分相似，`Pouch` 容器相较行业内的其他容器解决方案还是存在一些差异化特性的，例如 `LXCFS` 容器资源隔离、存储卷磁盘空间限额等。可以通过 `pouch run --help` 命令查看 `Pouch` 容器的可用操作和参数，如下所示。

```
$ sudo pouch --help
Usage:
```

pouch [command]

Available Commands:

create	使用指定镜像创建容器
exec	在运行的容器中执行命令
gen-doc	生成 Pouch 客户端文档
help	显示任意命令的详细帮助
image	管理本地镜像
images	列出所有的本地镜像
inspect	获取指定容器的详细信息
network	管理 Pouch 容器网络
pause	将指定容器暂停运行
ps	列出所有容器
pull	从镜像仓库获取镜像并缓存到本地
rename	重命名已创建的容器
rm	删除指定容器
rmi	删除指定镜像
run	创建并运行指定容器
start	启动尚未运行的指定容器
stop	停止运行中的指定容器
unpause	恢复已暂停的指定容器
version	展示 Pouch 的版本
volume	管理 Pouch 容器的卷存储

诚然，阿里巴巴并非首个在大规模生产环境下采用容器技术的企业，但却算得上首个将大规模生产环境下的传统业务全量迁移到容器中的企业级案例。在这其中，Pouch 容器功不可没，相信它的开源也将为行业带来更多的启发和思考。

8.7 微内核操作系统：Unikernel

8.7.1 Unikernel 概述

2016 年 1 月份，Docker 公司宣布收购微内核操作系统公司 Unikernel Systems，这条新闻在当年引起过相当大的轰动。虽然 Unikernel 随着时间再次逐渐淡出公众视野，但在本书的末尾还是要介绍一下这个思路十分独特的、古老而又先进的技术。

“Unikernel”的概念在早在 20 世纪 90 年代的时候就已经有雏形了，它是一类特殊操作系统的通称。最早的 Unikernel 操作系统原型是 Exokernel 和 Nemesis，之后相对比较出名

的有 Xen 社区主导的 MirageOS、前 KVM 开发者主导的 OSv 以及由 NetBSD 公司开源的 Rump Kernels。

在通常的操作系统中,系统的内核与用户的应用程序之间是有明确分界的,在 Windows 或是 Linux 中都有清晰的“内核态”和“用户态”定义,并通过“系统调用”的方式进行数据交换。这样做的目的是将操作系统与硬件打交道的功能与上层应用隔离,从而屏蔽其底层的差异性和复杂性。而在 Unikernel 的操作系统中是没有这种差别的,所有程序与底层的核心驱动都运行在一起,不存在运行时状态上下文切换的额外开销。直观来说就像是所有程序都运行在“内核态”(这种表述严格来说是不正确的,Unikernel 没有“内核”的概念),每个应用程序在编译时直接指定引入特定的驱动和核心 library,相当于每个程序都自带操作系统,而且是单独定制裁剪的操作系统。

通过 Unikernel 系统方式构建出来的应用程序都是独立发布且直接运行在虚拟化平台上的,这种系统原生不存在多用户和多任务的复杂场景,因此可以做得十分精巧。你可能会说,这不就是嵌入式操作系统吗?单纯从概念上看,Unikernel 与嵌入式系统颇有几分相似,但 Unikernel 系统的目标运行环境是虚拟化的基础设施,而不是那些嵌入式硬件设备。

图 8-13 展示了在 Unikernel 运行的服务与通过虚拟机以及容器运行的服务在结构上的差异。

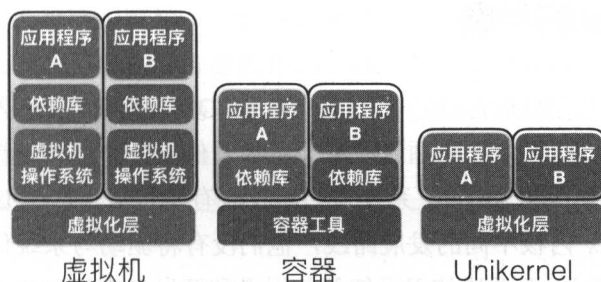


图 8-13 虚拟机、容器和 Unikernel 上运行的服务

Unikernel 的优势可以用两个词概括：架构简单、安全高效。

首先,Unikernel 抹去了现代操作系统由于软件层级抽象而导致的复杂性。越是通用的操作系统(比如 Linux 或 Windows)就包含了越多对特定的应用而言并不必要的服务、驱动、依赖包等。譬如 USB 驱动这类东西在虚拟化的云环境中其实是无用的,但在内核运行时仍然会将它加载进来。这些多余的内容既增加软件运行的负担,又额外消耗了非必需的资源。

其次,Unikernel 消除了运行时内核态与用户态切换的过程。Unikernel 的核心驱动与应

用程序是同时打包构建的，其内存是单地址空间（single address space），不区分系统内核区域和应用服务区域，因此不论是在启动速度还是在程序执行效率上都远高于通用的服务器操作系统。通常一个 Unikernel 系统的启动时间都在几十毫秒左右，可以这么说，只需一眨眼的功夫，成千上万个服务节点的集群就启动就绪了。

此外，Unikernel 带来的还有安全性方面的提升。一方面，只运行操作系统的核心，抛掉那些可能是漏洞来源的视频、USB 驱动、系统进程，极大地减小了可攻击的面积。另一方面，对于 Unikernel 而言，每一个操作系统都是定制用途的，其中包含的核心库均不相同，即使某个服务构建的操作系统由于特定软件问题而遭到入侵，同样的入侵手段在其他的服务中往往不能生效。这无形中增加了攻击者利用系统漏洞的成本。

最后，Unikernel 也是一种不可变的基础设施（Immutable Infrastructure）。不可变的基础设施的构想是 Chad Fowler 于 2013 年提出的，并随着 Docker 的流行而为人们熟知。Unikernel 系统一旦编译完整就不可改变，要想对其中的内容进行重新定制的唯一办法就是修改源码然后重新编译。这种软件实施方式有点像使用一次性产品，即安装一次，不做修改，用过即扔。这种方式对保障软件部署的一致性、提高运维效率和降低管理的复杂性方面带来的好处已经被许多容器化的实践所证明。

8.7.2 Unikernel 的发展

Unikernel 的发展可谓跌宕起伏，它的前身是 LibOS，后者源于嵌入式操作系统架构的一次研究性的尝试。LibOS 将操作系统应用于各种硬件设备的驱动进行抽象，形成了各种不同的可替换的 library，这与 Linux 系统最初的发展有些相似。然而 LibOS 的创造者们随后选择了一条与 Linux 内核不同的发展路线，他们没有将驱动与系统代码进行进一步的封装和整合而形成“内核”，而是让使用系统的使用者和开发者直接与驱动通信，只需要在编译时引入适当的 library，便能构建出快速适应不同硬件设施的应用服务。

随着 LibOS 的发展，许多各立门户的 LibOS 操作系统分支开始出现，这些系统通常都只具备用于特定硬件的专属驱动，并且由于开发者自身的喜好和偏向性，各系统对不同硬件的支持差异较大。而同一时期诞生的以 Unix 和 Linux 为代表的另一操作系统门派，凭借着更好的易用性和统一性逐渐占领了主导地位，这个门派也就是现在大家所熟悉的大而全的系统内核，多用户、多任务，加上严格的内核与用户分界的操作系统风格（要是当年胜出的是 LibOS，整个计算机行业的发展路线也许就完全不一样了）。

值得庆幸的是，在 LibOS 并未完全没落之前还发生了一件事情，那就是虚拟化技术的

兴起。随着虚拟化的广泛应用，一些 LibOS 系统也开始提供用于虚拟化基础设施的 library 驱动。正在 LibOS 逐渐势弱之际，当时的虚拟机化新秀 Xen 发布了一个独立的社区操作系统项目：MirageOS^①。这个项目是一个基于 LibOS 理念设计的开源操作系统，但它比其他的 LibOS 系统更进一步之处是，索性直接抛弃了所有物理硬件的驱动支持，专心一致做虚拟化，而将硬件的差异统统交给虚拟化平台去完成。同时，MirageOS 借着 Xen 虚拟化的广泛应用，打出了“Cloud Operating Systems”的大旗（因为 MirageOS 只能运行在虚拟化云平台上），并将这类专门运行在云环境的 LibOS 系统称为“Unikernel”，这个叫法一直沿用至今。

在 Unikernel 不断演进的过程中，分化出许多不同的发展方向，有些带有浓厚的学术色彩，有些则充满创造性或实用性。其中值得一说的是 Rump Kernels^②和 OSv^③，这两个项目是 Unikernel 对现有系统软件的兼容性改善方面做出的十分有价值的尝试。

MirageOS 系统是用 OCaml 语言编写的。作为当时典型的 Unikernel 程序运行方式，用户也必须通过 OCaml 编写自己的程序，这样才能在编译的时候和操作系统共同打包成可运行的软件。虽然 OCaml 是一种理念十分先进的函数式编程语言，但它最终没能成为主流，这注定了 MirageOS 操作系统很难在市场上有大的作为。有了前车之鉴，Rump Kernels 和 OSv 系统在设计时，分别通过各自的方式对现有的主流软件体系进行兼容。

Rump Kernels 的思路干脆明了，Unikernel 的架构针对的就是大规模的虚拟化服务器应用，既然 POSIX/Linux 已经是服务器软件的主流标准，那就兼容这套标准。Rump Kernels 系统核心使用 C 语言开发，提供与 POSIX 相关的各类 library 和符合 POSIX 标准的 C/C++ 编译器，又原生编译了包括 Python、Ruby、PHP、Bash、Nodejs、Rust 等诸多语言的解释器和编译工具。这样一来几乎绝大多数能在 Linux 上使用的软件都可以快速迁移到 Rump Kernels 上了。

OSv 系统同样也是使用 C 语言编写的，不过它要稍微任性一些，因为 OSv 的 C 编译器并不兼容 Linux 的 POSIX/GCC 标准。即便如此，许多标准的 C 语言应用（没有用到 POSIX/GCC 编译器特殊语法）都是可以直接在 OSv 编译运行的。此外，OSv 的高明之处在于，它成功地移植了 Java 虚拟机（这是 Rump Kernels 都没有做到的），这个在 TIOBE 编程语言流行度排行榜长居第一位的语言的确为 OSv 系统留住了不少用户。此外，OSv 还进一步提供了包括 Tomcat、Cassandra、Redis 等一些主流标准服务的官方系统镜像，省去了用户自行编译的麻烦。

① <https://mirage.io>

② <http://rumpkernel.org>

③ <http://osv.io>

2014 年，一部分 MirageOS 和 Rump Kernels 的开发者和拥护者从 Docker 的兴起看到了 Unikernel 系统的希望，它们发起了一个新的项目：IncludeOS^①。这个系统延续了 Rump Kernels 系统的思路，并制定了更积极的目标：全面支持 ISO C++11/14 标准编译器。2015 年底，这些开发者们共同成立了 Unikernel.org 社区，并注册 Unikernel Systems 公司（就是后来被 Docker 收购的那个公司）。

现在，Unikernel 系统似乎又从喧嚣转入低调和平静，继续在操作系统的发展史上扮演着“在野党”的角色，未来会怎样，谁知道呢？

8.7.3 体验 Unikernel

使用 Unikernel 运行的服务通常需要先和操作系统一起打包以后才能使用，因此如果在使用的時候修改了服务代码，就需要重做整个操作系统镜像。如果工程使用的是 C/C++ 这种静态编译型语言，由于通常用户不是在 Unikernel 系统上进行代码开发的，则需要先将源码重新编译。这实际上是在做交叉编译（在一种操作系统或体系结构，编译出另一种操作系统或体系结构的二进制文件），需要专用的编译工具链，如果是 Ruby、Python、Java 这类解释型的语言，则可以省掉这一步骤。然后才是将系统库和应用程序打包在一起，打包过程与制作 Docker 容器镜像有点类似。下面以构建 C/C++ 应用为例，分别讲解一下 Rump Kernels 和 OSv 系统的例子。

Rump Kernels 系统的工具链比较复杂，像是传统嵌入式开发的开发模式，不过好在有了 Docker 以后，可以把需要的工具软件预装到 Docker 镜像里，从而简化开发环境的准备。在 2015 年巴塞罗那的 DockerConEU 上，来自 Unikernel.org 社区的开发者 Justin Cormack 在现场演示了使用 Docker 镜像提供 Rump Kernels 编译环境，打包出能够在 KVM 上运行的 Nginx 程序的例子^②。

现在可以从 GitHub 上获取到当时演示使用的项目源码，如下所示。

```
$ git clone https://github.com/Unikernel-Systems/DockerConEU2015-demo.git
```

在 GitHub 有一个简单的说明文档，不过其中第一步的 `make pull` 操作需要下载的提供交叉编译工具的镜像有些已经不在 Docker Hub 上了，需要到这些镜像的源码仓库^③手动

① <http://includeos.org>

② <https://github.com/Unikernel-Systems/DockerConEU2015-demo>

③ <https://github.com/mato/rumprun-docker-builds>

构建出来。

然后在演示项目的目录中执行 `make` 命令，如下所示。

```
$ cd DockerConEU2015-demo
$ make
```

这个命令将依次进入当前演示项目中的 6 个子目录中，使用提供交叉编译工具的 Docker 镜像，在容器中分别构建出 MySQL、Nginx 和 PHP 的 Rump Kernels 系统镜像文件（命名为“mysql.bin”“nginx.bin”“php.bin”等），并将生成的 Rump Kernels 镜像和用来启动这些镜像的脚本一起分别打包成新的 Docker 镜像（命名为“unikernel/mysql”“unikernel/nginx”“unikernel/php”等）。

确保当前 Linux 主机上已经安装 KVM 和 Qemu 相关的包，接下来演示通过 KVM 虚拟化启动 Rump Kernels 镜像，如下所示。

```
$ make rundns
$ sudo ./docker-unikernel run -P --hostname nginx unikernel/nginx
```

`make rundns` 的命令在 Makefile 文件里面可以看到，实际上是通过 Docker 运行了一个 `mgood/resolvable` 镜像的容器，并将系统的“/etc/resolv.conf”文件挂载到容器里。这个容器提供了 Docker 的 DNS 解析的功能，可以用本地运行的容器的主机名作为域名，访问到相应的容器。

第二个命令中的“`./docker-unikernel`”脚本就在这个项目的根目录，它实际上还是运行了 `docker run`，只是在运行之前先配置了一个网络 Namespace，然后给容器的启动加上 `--device=/dev/kvm:/dev/kvm` 参数，最后对容器网络配置进行了修改。从结果上来说，就是用 `unikernel/nginx` 这个镜像启动了一个容器，并且给容器主机名设置成“nginx”。

`unikernel/nginx` 这个镜像做了什么呢？阅读项目“nginx”子目录里的 Dockerfile 文件不难发现，这个容器里包含了 Nginx 的 Rump Kernels 镜像以及“`run.sh`”这个脚本，并设置 CMD 属性为运行“`run.sh`”脚本。再来看“`run.sh`”脚本里的内容，最终执行的是这个 `qemu-system-x86_64` 命令，而脚本前面的部分都是在给最后这个命令拼装执行的参数。因此整个逻辑就清晰了：容器在启动后，会自动执行“`run.sh`”脚本，并在容器里面通过 KVM 工具将 Nginx 的 Rump Kernels 镜像作为虚拟机在当前环境下启动起来。

此时在当前主机上访问“`http://nginx`”地址，由于主机的“/etc/resolv.conf”文件已经被修改，会用“`mgood/resolvable`”镜像创建的容器作为 DNS 服务器，从而访问到了主机名是“nginx”的 Docker 容器，这个容器的网络 Namespace 被修改过，因此用户最终访问到的是在这个容器中创建的那个运行在 KVM 虚拟机上的 Nginx 服务。这个虚拟机的镜像

里的内容是已经与 Nginx 服务合为一体的 Rump Kernels 操作系统。

相比之下，OSv 操作系统的构建工具链则更加简捷，所有的功能都被封装到了一个叫 Capstan 工具里面，使用 Capstanfile 来描述构建的配置。Capstan 是一个使用 Go 语言编写的小工具，在它的 GitHub 仓库^①中提供了一个下载脚本，可以将 Capstan 的最新发布版本下载到用户的“\$HOME/bin”目录下，如下所示。

```
$ curl -sSL \
    https://github.com/cloudius-systems/capstan/raw/master/scripts/download \
    | bash
$ ls ${HOME}/bin
capstan
```

为了方便使用，建议将这个文件拷贝到 PATH 变量的目录中，如下所示。

```
$ sudo mv ${HOME}/bin/capstan /usr/local/bin/
```

如果本地安装有 Go 语言的 SDK，也可以使用仓库里的“install”这个脚本从源码的主干版本编译出最新版的 capstan 工具，如下所示。

```
$ curl -sSL https://github.com/cloudius-systems/capstan/raw/master/install | bash
```

Capstan 工具的命令风格模仿了 Docker，从表 8-6 的命令组合中就能看出。

表 8-6 Capstan 工具的命令及其说明

命 令	说 明
capstan run	启动虚拟机实例
capstan instances	列出虚拟机实例
capstan stop	停止虚拟机实例
capstan delete	删除虚拟机实例
capstan build	构建虚拟机镜像
capstan pull	从仓库下载虚拟机镜像
capstan images	列出虚拟机镜像
capstan info	显示虚拟机镜像详细信息
capstan import	从本地导入虚拟机镜像
capstan rmi	删除虚拟机镜像
capstan search	在仓库中搜索虚拟机镜像

例如，使用 capstan run 命令启动一个“cloudius/osv”镜像的 OSv 系统虚拟机，当

① <https://github.com/cloudius-systems/capstan>

Capstan 工具发现本地没有这个虚拟机镜像时，它会自动从官方仓库中下载，如下所示。

```
$ capstan run clouddius/osv
Downloading clouddius/osv/index.yaml...
170 B / 170 B [=====] 100.00 %
Downloading clouddius/osv/osv.qemu.gz...
21.62 MB / 21.62 MB [=====] 100.00 %
Created instance: clouddius-osv
OSv v0.24
eth0: 192.168.122.15
[/]%
```

在一个有 Capstanfile 文件目录里，使用 `capstan build` 命令就可以构建出新的虚拟机镜像，如下所示。

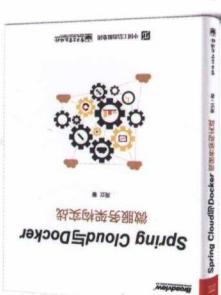
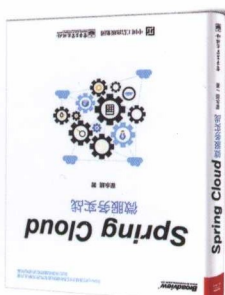
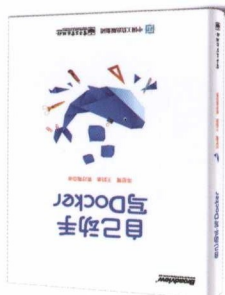
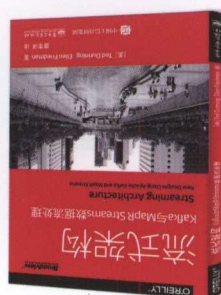
```
$ capstan build demo
Building demo...
Uploading files...
1 / 1 [=====] 100.00 %
```

8.8 本章小结

本章介绍了七种特色鲜明的容器周边技术。其中将容器作用发挥到极致的操作系统 Container Linux、基于容器架构的操作系统 RancherOS 是容器技术与操作系统结合的产物；容器式虚拟机 Hyper、虚拟机式容器 LXC 是容器技术与虚拟机技术结合的产物；企业级的容器工具 Pouch、无守护进程的轻量级容器 Rkt、将内核与应用合体的 Unikernel 则是由容器技术的而捧红的周边产品。

技术的融合总是能创造出新的机遇和惊喜，如今容器已经与诸多周边领域碰撞出了不少创意的火花。随着与容器相关的应用领域不断扩大，相信还将有更多这样的创意产品被设计出来，容器技术的未来值得期待。

好/书/分/享



微服务架构时代对我们的基础设施管理提出了非常大的挑战，DevOps和持续交付成为了基本能力要求。没有近两年容器技术的快速发展，这些能力可能仍然只属于少数技术实力强大的互联网公司。容器技术的应用从真正意义上催化了我们在基础设施管理上的革命，成为时下每个IT组织必须获取的能力。

本书就像一个容器，承载了容器技术领域的方方面面，极大地解决了大家面对层出不穷的容器技术时的不知所措。林帆作为这个领域的专家，通过这样一个“容器”减轻了我们入门学习的烦琐，同时也给出了很多中肯的建议，避免读者在探索过程中走不必要的弯路。

肖然 ThoughtWorks中国区咨询团队总监

《容器即服务》这样的标题，喻示着容器已然成为云服务的一种形式，而事实上也是如此，在Docker问世之初，Google GKE和AWS ECS就将容器带到了云上，而随着时间推移到2017年的下半年，微软的ACI和AWS的Fargate更是扶正了容器在云计算领域中一等公民的地位。看着林帆同学的目录，回想起过去三年间自己也真实地参与了这场容器与云的进化过程，感慨万千。我和林帆认识大概有两年了，印象里，他一直是一个认真而敏锐的人，容器技术领域发生的一切都逃不出他的视野。任何一项技术走向繁荣，都离不开技术作者们的辛苦工作，非常感谢林帆的这份付出，也期望本书能帮助更多人了解并使用容器，乃至参与到容器相关的开发工作中来。如果你想全面了解集群与容器编排领域，相信这本书可以给你足够的信息量。对于想更进一步深入到容器领域中的读者，我建议你更仔细地关注Kubernetes的章节，而如果你对Hyper这样的底层技术有兴趣的话，同样欢迎在读完本书相关章节后来和我们交流。

王旭 Hyper创始人兼CTO

本书深入浅出，对企业级容器化技术落地实践涉及的主要方面都做了翔实的分析及介绍，作者有多年的企业级微服务及容器化改造实践经验，本书是不可多得的容器技术进阶读物。

秦小康 RancherLabs大中华区总经理

林帆老师是斯达克学院（StuQ）的明星讲师，其主讲的《Docker容器集群技术》课程影响和帮助了近500余位学员，其中由浅入深地讲解了容器集群技术从理论到生产环境的实战技能，其专业素养和敬业精神获得了五星好评。林老师通过学员的课堂问题和反馈也收到许多宝贵意见和补充素材，使得本书更加贴合工程技术人员的需求。非常高兴看到本书的出版，它将帮助各位学习容器技术解决方案的同学找到方向。

雷蒙德 斯达克学院（StuQ）业务总监兼主编



博文视点Broadview



@博文视点Broadview



策划编辑：张春雨
责任编辑：牛勇
封面设计：李玲

上架建议：软件开发 / 架构

ISBN 978-7-121-33276-0



9 787121 332760 >

定价：99.00元